

AD-A223 074



CECOM

CENTER FOR SOFTWARE ENGINEERING
ADVANCED SOFTWARE TECHNOLOGY

Subject: **Final Report - The Software First System
Development Methodology**

DTIC
ELECTRIC
JUN 21 1990
S E

CIN: C01 09100 0001

15 FEBRUARY 1989

CLEARED
FOR OPEN PUBLICATION

MAY 2 - 1990

This document has been approved
for public release and sale; its
distribution is unlimited.

DIRECTORATE FOR PROTECTION OF INFORMATION
AND SECURITY - NEW (OASD-P&S)
DEPARTMENT OF DEFENSE

90 002030



Accession For	
NTIS	<input checked="" type="checkbox"/>
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution _____	
Availability _____	
Dist	Availability Special

FINAL REPORT

THE SOFTWARE FIRST
SYSTEM DEVELOPMENT METHODOLOGY

CONTRACT NUMBER: F030602-86-C-0111

IITRI PROJECT NUMBER: A06226

PREPARED FOR:

U.S. ARMY, CECOM
ADVANCED SOFTWARE TECHNOLOGY
AMSEL-RD-SE-AST-SS-R
FT. MONMOUTH, NJ 07703-5000

PREPARED BY:

DAVID PRESTON
ELAINE FEDCHAK
IIT RESEARCH INSTITUTE
4600 FORBES BLVD.
LANHAM, MD 20706

JANUARY 1989

TABLE OF CONTENTS

	<u>Page</u>
1.0 INTRODUCTION	1
2.0 PHILOSOPHY OF SOFTWARE FIRST	2
3.0 BACKGROUND AND RATIONALE OF SOFTWARE FIRST	6
3.1 Background of the Software First Concept	6
3.2 Rationale	8
4.0 MODELS OF SOFTWARE FIRST IDEAL AND REVISED	13
4.1 Ideal Model	13
4.2 Revised Model Requirements Definition and Allocation	15
4.3 Revised Model System Development and Integration	20
4.4 Interim Software First System Development Model	23
5.0 AREAS FOR SPECIAL CONSIDERATION	26
5.1 Requirements Definition	26
5.2 Performance Evaluation	32
6.0 DETAILED DISCUSSION OF SELECTED TECHNICAL APPROACHES	34
6.1 Prototyping	34
6.1.1 The Motivation for Prototyping	34
6.1.2 Paper Prototypes	35
6.1.3 Nonfunctional Prototypes	36
6.1.4 Working Prototypes	37
6.1.5 Development Prototypes	37
6.1.6 Prototyping Tools	38
6.2 Calibrating Software On the Host Environment	38
6.2.1 Simulating the Target Environment	39
6.2.2 Identifying Bottlenecks	42
6.2.3 Optimizing Memory Management	45
6.2.4 Objectives of Calibration	46
6.3 Portability	47
6.3.1 Definition of Portability	48
6.3.2 Benefits of Portability	48
6.3.3 Challenges to Portability	50
6.3.4 Approaches to Improve Portability	52
6.3.5 Issues	53
6.3.6 Measuring Portability	55
6.4 Establishing Acceptance Criteria	55
6.4.1 Importance of the Acceptance Function to Software Development	55
6.4.2 Traditional Approach to Acceptance	56
6.4.3 The Acceptance Function in the Software First Approach	56
6.4.5 Specific Challenges	58
6.5 Graphically Oriented Techniques	58

7.0	USE OF COMPONENTS OF EXISTING METHODOLOGIES AND TOOLS	61
7.1	Reusing Components of Existing Methodologies	61
7.2	Tools	65
8.0	REFERENCES	73

LIST OF FIGURES

Figures

3-1. The Too Often Result of Traditional Development	8
4-1. The Ideal Model of Software First	13
4-2. Requirements Definition and Allocation	16
4-3. System Development and Integration Activities	20
4-4. Interim Software First System Development Model	23
6-1. Portability/Performance Curve	54

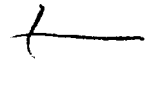
1.0 INTRODUCTION

Department of Defense systems are becoming increasingly dependent on software. A decade and a half ago software costs began to exceed hardware costs. Problems uncovered very late in the development cycle of major weapons systems are frequently described as software problems. Still, these systems are being developed by thinking of the hardware for the system first; by deriving requirements based on hardware capabilities; by selecting hardware and designing the software to be compatible with it; and, finally, by correcting any shortcomings in the hardware with clever software fixes. The Software First System Development Methodology proposes to place the emphasis on the software.

The major benefits to developing the software for a system prior to hardware selection include the development of software that will be more maintainable because all design decisions will be based on what is best for the software; the selection of hardware after the software has been developed, which enables a wiser choice because more is known about the system, as well as permitting the hardware to mature an additional few years while the software is being developed; and the better management of evolutionary requirements because no hardware has been selected and therefore locked into the system.

This report is an iteration of the ongoing definition of software first. Section 2 provides the philosophy of software first. Section 3 describes the history of the phrase "software first," which provides the historical perspective for the methodology being developed in this research effort. Section 4 provides a description of software first from the top-down perspective. This section details the goals and principle objectives of this methodology. Section 5 identifies activities that are central to the successful implementation of software first and warrant special consideration. Section 6 details selected technical approaches to meeting the challenges of software first. Section 7 takes a bottom-up approach to software first by investigating the potential for reusing components of existing methodologies and using existing tools.

2.0 PHILOSOPHY OF SOFTWARE FIRST

Software first is a new approach to system development. The software first system development methodology provides a means to alleviate problems associated with traditional development methodologies. The basic tenet of the software first approach is to postpone selection of the target hardware until as late as possible in a system's development. The major technological advancement that makes the software first approach feasible is the Ada language. (KR) 

System development is a complex undertaking. One layer of complexity is driven by the many points of view that interact in a system development:

- o eventual user
- o customer (the user's agent; may be the user)
- o system developer
- o hardware developer (and/or vendor)
- o software developer
- o tester
- o maintainer
- o logistician
- o managers from the various management disciplines.

Each point of view generates a different set of priorities and goals. Such multiple viewpoints and interests add to the inherent technical complexity of modern systems. Particularly for CECOM, systems tend to be large, embedded, have real-time requirements, be long-lived, expensive, critical, and have high reliability, redundancy, and security requirements. Each of these factors adds to the complexity of system development.

Complexity leads to difficulty. System development is hard. As computer science evolves, especially software engineering, many approaches to attacking the known difficulties in system development have been proposed and tried; an ad hoc methodology has evolved. But the difficulties remain.

No cure-all has been found that is applicable to all systems. The task is too complex.

This ad hoc methodology is called the "traditional approach" in this paper. It generally follows a sequence of development phases:

- o system requirements analysis
- o partitioning into software and hardware requirements
- o selection/purchase/design of hardware
- o preliminary and detailed design of software
- o implementation: hardware fabrication, software coding
- o software testing and hardware testing (independently)
- o hardware and software integration and testing
- o system acceptance, fielding, and user training.

Even though the traditional approach has known weaknesses, time and budget constraints often get in the way of changing it during actual system development projects. So even though we know that a system's requirements should be clear, consistent, and complete before we design and build it, we often jump ahead while the requirements are still being solidified. We are afraid that if we invest the time and effort needed to iterate the requirements until we are sure they are right, we will expend all of the time that was available for the whole project; thus leaving no time to do the design and implementation. The current tendency is, therefore, to start in on the design and implementation, hoping that 1) the requirements are "close enough" and well enough understood, and 2) the design and implementation processes will uncover any missing or inconsistent requirements, as well as further the developer's understanding of the requirements.

A solution would be to develop better ways to do requirements analysis. Then the requirements would be right, complete, consistent, and understandable to both the user and developer early in the project development; thus leaving sufficient time and resources to do the

implementation. The proposed software first approach emphasizes requirements definition for this reason.

Similarly, even though we know that the software for a system is more easily maintained if it is modular, straightforward, understandable, and written using a high order language and self-documented code, we override these quality requirements when trying to fit the software into a particular target architecture or onto a specific processor. We trade away modularity for speed. We forgo portable Ada constructs in favor of machine-dependent timing sequences. A solution would be to delay the selection of hardware until the software has been developed. That way, the needs of the software would come first. Hardware would be selected that fits the software, rather than having to compromise software engineering principles.

The major implication of the software first approach is that the software development will become the driver in a system development. If the software first approach is implemented correctly, the software can outlive several generations of hardware. It therefore makes sense to invest time, effort, and other resources in developing high quality (e.g., portable, maintainable, reliable) software because of the extended system life.

Taken further, the software first approach leads to highly modular software, with hardware dependencies and software functions encapsulated in Ada packages. Over the lifetime of the system, hardware may become available that performs some of the functions originally allocated to software. It will be a straightforward operation to replace the software bodies with interfaces to new hardware.

Software is abstract and infinitely flexible, so it has traditionally been asked to bear the burden of changes when an integration runs into problems. Recent software engineering experience has shown, however, that unanticipated software fixes have a detrimental effect on software quality and therefore overall system quality. A better approach is being sought. Hardware is cheaper to buy than software is to custom build. Hardware is evolving rapidly, often faster than the software can be developed to run on

it; therefore, it makes sense to focus on the software first and choose the hardware to fit later.

The software first approach may seem radical, especially to many system developers. It requires a new way of organizing a system development and presents some technical challenges. But traditional approaches have enough shortcomings to make investigation of a possibly better way worthwhile. The remainder of this report proposes and discusses an approach to system development based on a software first philosophy.

In our investigation of the software first concept, we are examining a broad spectrum of relevant issues to determine how the software first approach can be implemented to optimize the system development process.

3.0 BACKGROUND AND RATIONALE OF SOFTWARE FIRST

3.1 Background of the Software First Concept

By the early 1970s it had been determined that software had become "the tall pole in the tent." The Air Force budget for fiscal year 1972 indicated that between \$1 billion and \$1.5 billion was spent on software, approximately three times the cost of hardware for the same period [BOEH73]. Not only was software costing more than the hardware component of a system, but it was also believed to be more responsible for schedule slippages, cost overruns, operational penalties, and performance penalties, as well as for complex embedded problems that surfaced after system fielding. In the one and a half decades since 1972, the problem has continued to grow and the software pole has become even taller.

Both the concept of software first and a software first development machine were proposed in the early 1970s, although both the concept and machine being proposed here are significantly different from their predecessors. The intent to bring system cost, schedule, and performance under control is still the same, but now the approach can be more ambitious. Although many of the problems highlighted earlier either remain or have grown in magnitude, a significant new tool exists that we believe can facilitate software first becoming a reality. That tool is the Ada programming language.

The software first concept of the early 1970s was not literally a software first approach but rather a concurrent software and hardware development approach [FLEI74]. The approach called for an iterative system design process with several iterations possible between logical levels. Once the design had been completed, software development and hardware fabrication could be initiated. The theory proposed that final testing needed to be done on the actual system hardware; therefore, the period for parallel development of hardware and software was between the completion of the system design process and final testing. Our proposed software first

concept is much more literal in its interpretation of the software development being first.

The software first machine proposed in the early 1970s was a generalized computer capable of simulating several computers and several computer configurations [BOEH73]. Software would be developed by assuming a specific architecture and then determining its adequacy as the software was developed. This would require that the software first machine be able to emulate the particular target machine of interest. Although various aspects of the architecture could be altered during the course of development (e.g., memory size and clock speed), the instruction set in which the software was being developed would obviously be fixed. The set of hardware options would then be limited to machines with the same instruction set. Our proposed software first development methodology would not require the construction of a specific software first machine with simulation capabilities. Rather, any computer with Ada cross compilers would be a candidate host machine, with the potential targets being the set of computers for which cross compilers from the host exist.

Although the software first machine of the 1970s is more consistent with what is being proposed here than is the software concept of the 1970s, there are significant differences with the machines as well. Rather than a special purpose machine being constructed for software development, the focus would be on a machine with strong software development tools and Ada compilers with several potential targets. The technology exists and has been successfully demonstrated to have multiple Ada cross compilers utilize the same front end [DEBA86]. The advantages of this technology include having a consistent user interface, having compiler enhancements available for all cross compilers at the same time, and making the development of new cross compilers more feasible. The significance of Ada's portability in general, and this cross compiler technology in particular, is that software first machines already exist that, given today's technology, make software first a quite plausible approach to system development.

3.2 Rationale

The traditional method of system development is to choose the hardware for the system, fit the software to the system, then add hardware components and both add and alter software components to make the system work. By the time the system is fielded, the hardware is several years old and no longer state of the art. The software design has been altered to fit the hardware of choice and has therefore become both machine-dependent and difficult to maintain. The schedule has slipped because of incompatibilities between the hardware and software. The negative effects of this approach are most pronounced during the maintenance phase of the life cycle. This traditional development leads to the life cycle depicted in Figure 3-1.

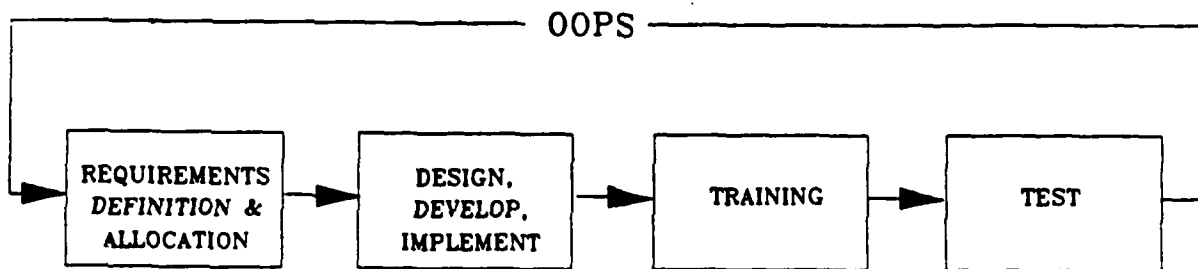


Figure 3-1. The Too Often Result of Traditional Development

In addition to out-of-date hardware and software that has lost its design integrity, the traditional development approach has other shortcomings. The software first approach has components that address these identified shortcomings. It should be noted that several of the components of software first do not require a strict adherence to the software first philosophy. Portions of this methodology will, therefore, be usable with a more traditional development approach; other portions will facilitate software first, but will not be critical to the software first methodology; and other portions will be specifically linked to this methodology.

The first component to be discussed is user involvement. The "user" in this context is the person who has the need for a computer system. A major cause of the "OOPS" in Figure 3-1 is that by the time the system has been developed, it no longer meets the needs of the user. This is because of several contributing factors that will be addressed specifically in the following paragraphs. The general cause, however, is the lack of interaction with the user during system development. The software first approach incorporates continuous user involvement during development.

Three specific areas of increased user involvement are requirements definition, training for the operator of the fielded system, and the man-machine interface. The purpose of requirements definition is to assure that the user and the developer, and any other key people involved in the development, mutually understand the requirements. Because English is an ambiguous language, requirements written in English are apt to be ambiguous; therefore, it is necessary to iterate the requirements to assure that they are mutually understood. Because complete mutual understanding and agreement is a naive goal, the user and developer must continue to interact during the development to bring to the surface differences in interpretation as early as they are recognized.

Training is another issue that involves ongoing user communication, as well as communication with the training organization if that organization is separate from the user. Software first encourages early training for multiple reasons. First, early training will identify any discrepancies between the skills necessary to operate the system and the abilities of the anticipated operators. Also, by utilizing a trainer, another perspective on the functionality of the system will be obtained that may further identify differences between what the user needs and what the developer is building. Further, with training started early in the life cycle, trained operators will be available when the system is fielded.

The man-machine interface is being stressed in this methodology for several reasons. First, it establishes a concrete view of the system's functionality. Next, it defines precisely what the human operator will have

to do and what information the system will present to him. Further, it provides a vehicle for beginning early operator training.

The establishment of a concrete view of the system's functionality provides the user and the developer with another opportunity to determine that they mutually understand the system requirements. Typically, a system's functionality is viewed from the global perspective, which tends to be abstract. This global perspective usually is expressed in terms of the effect of the system on its environment. By defining the man-machine interface, the user and developer can observe exactly how the system will implement the desired effect. This more concrete view of how the system will operate increases the probability that the user and developer truly understand each other.

One of the more common causes of the large "OOPS" in Figure 3-1 is the development of a system that the intended operator is unable to operate. This occurs when the developed man-machine interface is incomprehensible to the human who is expected to operate the system. The traditional adjustment is to build another layer onto the existing man-machine interface to enhance the system's ease of operation. This additional layer has several negative effects: it degrades performance, it alters the software architecture, it increases cost and delays schedule, and generally aggravates a bad situation. By defining what the operator will have to do early in the life cycle, necessary adjustments can be made. The user and developer can rationally select the best of the available alternatives instead of being forced into making a decision under the pressures of a delayed, overrun delivery.

So far the rationale for this methodology has focused on several factors: increased user interaction, improved requirements definition, enhanced training, and early development of the man-machine interface. The single most important rationale, however, is that it allows a delay in the selection of system hardware.

The software first system development methodology makes an explicit distinction between the hardware used to perform some system function and the hardware used to execute the system software. Advances in both types of hardware make the postponement of hardware selection attractive. Developing the software prior to hardware selection may enable the system developer to utilize hardware that is five or 10 years more advanced than hardware that is available at the initiation of development.

Another driving force in the development of software first is the need to control and reduce system maintenance costs. Software maintenance costs can dominate software costs, consuming from 40% to 80% of the total life-cycle expenditures. This is due, in part, to adjusting the fielded software to what the user needed in the first place, to updating the hardware in a recently fielded system, and to altering the man-machine interface so that the operator can operate the system. Each of these concerns has already been addressed in this rationale. The other major contributing factor of high maintenance costs is that, traditionally, all maintenance activities are performed on the target hardware. With software first, however, maintenance activities are performed in the more robust and supportive host environment. When a change to the fielded system is necessary, to either correct errant operation or to introduce a new capability, the change will be implemented in the host environment. Use of the software tools available on the host will greatly reduce the effort required to update the code and documentation and significantly reduce the cost of retesting the system.

The intent of this methodology is to allow software to be developed prior to selection of the target computer; to permit development and initial testing on a programmer-friendly host computer; to determine functional and operational software changes by allowing end users to execute the software on the host; to implement the necessary changes on the host system; to define the hardware requirements by instrumenting the host system; and to enhance the confidence in the software component of the system prior to porting the software to the target hardware. The benefits of this approach will hopefully be realized both during development and, more importantly, during the operations phase of the system life cycle. When changes are

proposed on a fielded system, their feasibility and impact can be assessed using the host environment prior to updating the targeted version of the system. Beyond the technical effects on system development, this approach will hopefully introduce more competition into the contracting process because intimate knowledge of a system-specific target hardware configuration will no longer be necessary to develop or maintain the software. The other anticipated major effect will be on training. Training will be started earlier, and lessons learned from the training process will be fed into the development process.

With the development and standardization of Ada, Department of Defense software has become more portable and less machine dependent; therefore, the software first system development methodology now appears feasible. It is assumed here that the portability of Ada will minimize the effort required to retarget the software developed. It is also assumed that any changes that must be made will be made to the software running in the host environment. Further, the development tools will be available at the site that performs the post-deployment software support. Finally, with decisions being made with respect to quality software engineering practices, requirements will be more easily maintained and traced and "requirements creep" minimized.

4.0 MODELS OF SOFTWARE FIRST IDEAL AND REVISED

4.1 Ideal Model

The ideal model of the software first system development methodology has been developed to define the desired attributes of the system in an ideal world. This is to ensure that the goals of the methodology are clearly articulated. Some of the goals are beyond the reach of today's technology; however, with these goals clearly stated, as technology matures, so can the methodology. Further, if the underlying goals are understood, when deviations from the ideal model are necessary, these deviations can be made with a clear understanding of what is being sacrificed.

The two major factors of the ideal model are that the user is heavily involved throughout the development cycle and that all hardware decisions are delayed until after software development. The delay in hardware selection supports that all development, training, and maintenance are done on the host. This delay also supports using the software running in the host environment to define the hardware needs for the system.

Figure 4-1 shows the ideal model for software first.

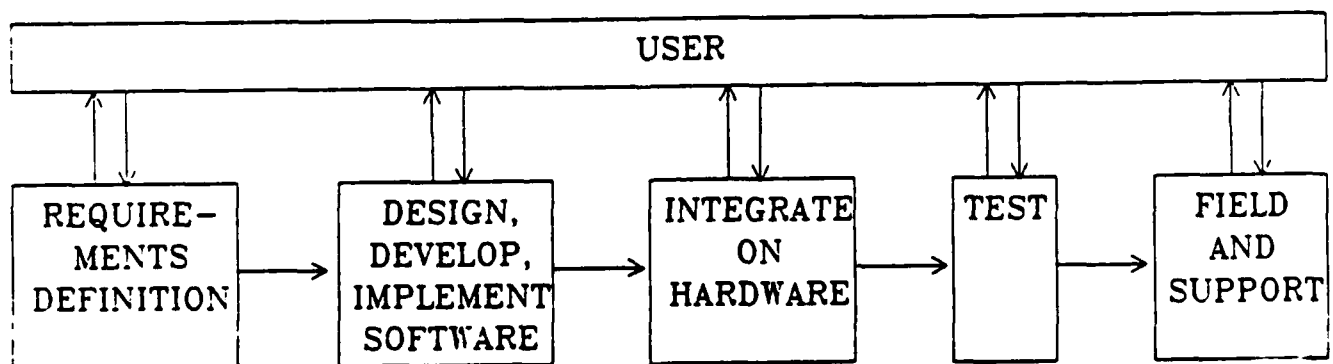


Figure 4-1. The Ideal Model of Software First.

The arrows in Figure 4-1 clearly indicate the emphasis placed on user interaction throughout the software first development cycle. Although the user interaction is continuous throughout the model, it is of particular importance during the requirements definition phase of the model. During requirements definition, requirements will be analyzed to assure that the following all occur:

The user is asking for precisely what he needs.
The developer understands what the user is asking.
The requirements are feasible with today's technology.

The second box in Figure 4-1 indicates that the software is designed, developed, implemented, and that it is implicit that this is to take place on the host environment. This is to ensure that the software portion of the system functions as intended on the host environment prior to selection of, and retargeting to, the target environment. Although the word "test" appears in a later box in the diagram, the software testing takes place as part of the implementation. Also implicit, though not on the diagram, is that all training and maintenance are to be performed on the host environment.

In this ideal model, the portability of Ada is assumed to be virtually 100%; therefore, the integration onto hardware is shown as a single, straightforward operation. The target machine is chosen based upon the needs of the software. While operating on the host, the memory and timing needs of the software are evaluated. Hardware is then selected that can meet these needs. The hardware that is embedded into the system to perform specific system functions is also chosen at this time. By delaying all hardware selections until after the software is developed, the most current hardware can be used in the system.

The "test" box is for acceptance testing, and at this point the adequacy of the hardware is assessed. If the initial hardware does not satisfy the system requirements, then a change in hardware must be made. It

is likely, however, that the hardware will be acceptable because the precise needs of the hardware were identified prior to its selection.

The primary significance of the field and support phase is that the support is performed in the host environment. When a suggested change to system software is made, the impact of that change on the other system software is assessed using the documentation. The change is implemented in the supportive host environment, instead of the austere target environment, and the entire system is retested to assure that changes made have not had an undesirable effect on the system. Of particular interest are potential effects on the system's ability to meet timing requirements.

This ideal model makes the following naive assumptions:

Once requirements definition is complete, the user and developer will completely understand each other and the system requirements and no ambiguity will exist in the requirements documents

A system can be designed and implemented without consideration of the capabilities of hardware

Ada code is completely transportable; therefore, regardless of the host used or target selected a cross compiler will exist to retarget the software without significant problems

Software executing on a host can be evaluated to completely define the requirements necessary for a target environment

Software functionality cannot be adversely affected by the selection of hardware chosen to implement specific system functions.

The purpose of the ideal model is to define what would be desirable in an ideal world. The next several sections of this report describe adjustments made to the ideal model to account for some real-world complications.

4.2 Revised Model Requirements Definition and Allocation

Requirements definition has traditionally been a problem with system development. The intent of requirements definition in software first, as was stated earlier, is to ensure that the user is asking for what he needs,

that the user and developer understand each other, and that the requirements are feasible given today's technology. An underlying theme of the approach used here is that the requirements focus on what is to be done, not how it is to be done. As Figure 4-2 indicates, only after the requirements have been established are they partitioned into software and hardware requirements. This will facilitate maintaining a focus on the requirements and avoid performing premature system design.

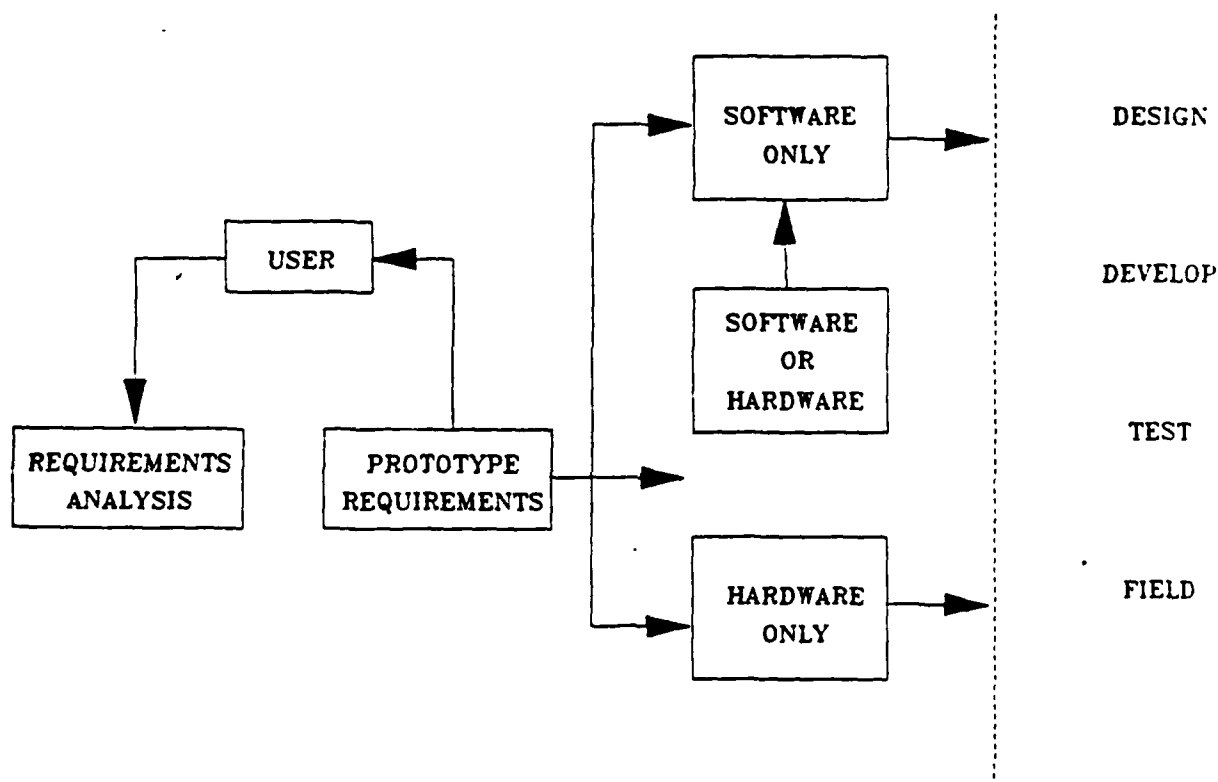


Figure 4-2. Requirements Definition and Allocation.

Three specific elements of the software first approach to requirements definition must be stressed: the establishment of "capability subsets," the interplay between Ada and requirements partitioning, and the bias toward implementing system functions in software. Although these will be presented separately, they are interdependent and the collective effect of all three is far greater than the sum of their individual effects.

At one level of abstraction, the intent of requirements definition is to ensure that the user and developer understand each other. At a more concrete level, the requirements definition phase is to develop a precise and concise statement of the requirements. This not only provides the opportunity for the requirements to be explicitly understood and to have their feasibility established, it also enables the establishment of "capability subsets."

The concept of capability subsets is not intrinsic to software first, but is a concept that is mutually supportive with software first. In brief, capability subsets are the groups of system capabilities that are independent and can be individually introduced into the system. This requires a much more detailed understanding of the system requirements so that related, but independent, requirements can be identified. Further, the process requires decomposing system requirements into components so that a requirement can be partially implemented, with the remainder of the requirement incrementally added to the system's capabilities on subsequent releases.

This process of developing capability subsets is meant to further examine, not to further detail, the system requirements. The intent of this examination process is to help identify related requirements. This will lead to a better structured requirements document, which will then drive a more structured design, and, ultimately, a more maintainable system. The process necessitates more emphasis on this early phase of development, which is intended to structure the requirements document based on interrelationships and interdependencies of the requirements. The result of this structuring is a better understanding of the requirements, both individually and collectively.

With the requirements grouped based on their technical or operational interdependencies, the user will be better able to establish priorities for the groups of requirements. When structured into groups of requirements that need to be considered and developed collectively, the user will be better able to focus on what system functions are necessary. This will lead

to the identification of the critical kernel or primary functions of the system. Once this kernel has been identified, the other groups of requirements can be prioritized. This may lead to a series of builds, with the series being planned so that development, testing, and fielding of increments can all be driven by a rational process based on related requirements and prioritized groups of requirements.

Other important outgrowths of capability subsets include the potential for identifying a group of requirements that can be implemented using existing components. In this way, a portion of a system can be fielded very quickly to partially meet the system need, with subsequent releases eventually completing the system requirements. Also, this increased examination of system requirements will assist in reducing the amount of alteration of requirements as the system is developed and deployed. It will restrict the evolution of a requirement to the level of changing to meet more detailed needs and eliminate or drastically reduce the need to add requirements based on unanticipated major system needs.

Capability subsets will be an outgrowth of a heightened requirements definition process. There will be systems, however, for which the establishment of capability subsets is neither technically nor economically feasible. Even in these cases, however, the enhanced requirements definition process will be performed.

The second key component in the requirements definition process is the Ada language. Of particular interest is the Ada package. The package makes two significant contributions to requirements definition in software first. First, a package may define a capability that will not be implemented in the initial delivery of the system. Second, a package may ultimately be developed in either software or hardware.

The advantage of putting specific capabilities into packages is that the package is built into the design as part of the initial design process, even if the package is not implemented as part of the initial delivery. Some accommodations must be made, obviously, to ensure that a package that

is not implemented is not called by the implemented system. Given the nature of Ada program design languages (PDLs) , the interdependencies of packages are quite easily observed and controlled.

The second contribution directly supports the central theme of software first, postponing the selection of hardware as long as possible. When requirements are being partitioned into software requirements and hardware requirements, the Ada package permits making assumptions about future hardware capabilities. If no hardware exists to implement a specific system requirement, but it is anticipated that such hardware will become available during the time that the system is under development, that requirement can be assigned to hardware. In the system design, that requirement can become a package, with the intent of implementing that package in hardware. If the hardware to implement the function does not exist when it is needed, the package can be implemented in software. Hardware developments can therefore be anticipated. If they do not materialize, the effect on the system is negligible.

The preceding paragraph notwithstanding, the bias of software first is to implement functions in software. Once system requirements have been developed and are being partitioned into hardware and software, all functions are to be implemented in software unless hardware is clearly superior for implementing the function. The rationale for this is to avoid the necessity of hardware upgrades, and the resulting impact on the system. With the degree of portability of Ada, changing the hardware that executes the software will not have a severe impact on the system. Changing the hardware that implements specific system functions, however, may cause perturbations in the system design. This can possibly be avoided by implementing those functions in software.

These three components, capability subsets, Ada packages, and a software bias, are interrelated. Individual elements of capability subsets will be implemented with Ada packages and, preferably, developed in software. To realize the anticipated benefits of this requires an enhanced

requirements definition. As software first evolves, there will be a continuing emphasis on requirements analysis.

4.3 Revised Model System Development and Integration

There are several key themes to the development and integration components of software first:

The major development, training and maintenance activities are all performed in the host environment.

Training begins early in the life cycle, and lessons learned in training relative to the system are fed back into the system development.

A system prototype is used to identify requirements that drive hardware problems.

A man-machine interface prototype is developed with extensive user involvement.

The software development process interacts with hardware selection and hardware development throughout the development life cycle.

Figure 4-3 shows the major components and their interactions for the development and integration activities for software first.

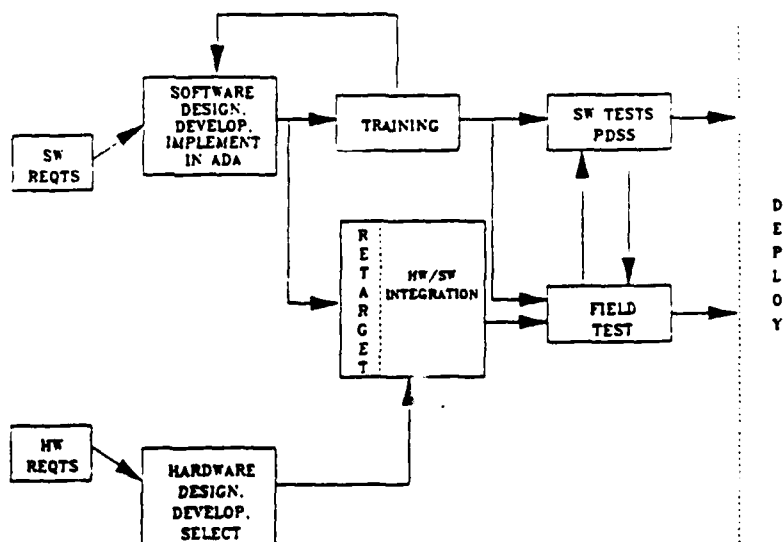


Figure 4-3. System Development and Integration Activities

In Figure 4-3, all activities along the top path are performed in the host environment. The design, development, and implementation of the software is performed on the host in a traditional development; the uniqueness of software first is that training and post-deployment support are also performed on the host. Training will be discussed in a later paragraph. There are several advantages of performing post-deployment support, or maintenance, on the host.

Typically, host environments have access to several sophisticated software development tools. These range from requirements analysis tools, to design tools, to documentation generation tools, to testing tools. These tools are not available in the target environment where maintenance is typically performed. The benefits of using the host for maintenance is twofold. The tools available on the host make updating the software easier, and the total impact of any changes to the software is easier to assess because the changes are being implemented in the same environment used for development.

As was mentioned earlier, the host will also be used for training. This is not the only change to the traditional development cycle relative to training that software first introduces. The changes are interdependent and are centered on the concept of performing the training on the host. The primary changes are that training will be performed earlier in the life cycle and that knowledge gained through the training process will be fed back into the development process. These changes mandate other changes.

First, in order to begin training early, a prototype of the man-machine interface must exist early. This is required to define exactly the set of functions that the operator of the system must perform and to detail the information presented to the operator by the system. If this early information indicates that the man-machine interface is inconsistent with the capabilities of the operator, then changes can be incorporated into the system.

Training on the host provides other advantages. It provides the opportunity to initiate training before the system has been fielded. This presents the opportunity to have the operators trained prior to system fielding. Also, because maintenance is also performed on the host, the updated system can be introduced to the operators prior to being fielded. This permits the opportunity to determine the effect on the operator of any system upgrades before those upgrades are released. Finally, the host environment is much more robust than the fielded system. Tools that are used for development can also be used to support the training process.

In addition to the training advantages that a prototype of the man-machine interface provides, the prototype enables the user to get yet another view of the developer's perception of the system. As the developer prototypes the man-machine interface, he must interpret the system requirements and reflect how these requirements will be presented to the system operator. This provides another concrete view of the abstract system requirements, enabling the user and developer to gain additional clarity on the system requirements. As these requirements gain clarity, the user and developer will be better able to determine and correct any discrepancies between their interpretations.

In addition to the man-machine interface, specific system requirements will be prototyped. The requirements chosen to be prototyped are those that are the most likely candidates to drive hardware problems. These requirements will be prototyped to determine the feasibility of effectively implementing them in the fielded system. The results of the prototyping effort will guide the decision to either implement the requirements as stated; modify the requirements to enhance their feasibility; implement a portion of the requirements now and defer a decision on the remaining portion; or eliminate particular requirements because of a lack of confidence that they can be successfully implemented in a cost-effective manner. Obviously, these decisions will be made jointly by the developer and the user.

The system prototype effort will focus on the requirements that are considered "high risk," and the connection between these selected requirements and any hardware necessary to implement them will be continuously explored. The more general hardware requirements of the evolving software will also be monitored and evaluated. The software first approach requires the software to be developed before the hardware is selected. In practice, it will be necessary to estimate the system's hardware needs early in the development cycle, and assess the feasibility of available hardware to meet these needs. As the software is being developed, the requirements for the hardware that will execute the software will become more clear. As these requirements become clear, the feasibility of existing hardware to satisfy the requirements will be assessed. In this way, the desires of the software world will be tempered by the realities of the hardware world.

4.4 Interim Software First System Development Model

Figure 4-4 is the interim model for the software first development methodology. This model is largely a union of the figures presented in the previous two subsections. Each of the previous drawings covered only a portion of the development cycle. Although most of the significant points about this interim model were made in the previous two subsections, there are additional points to be made here.

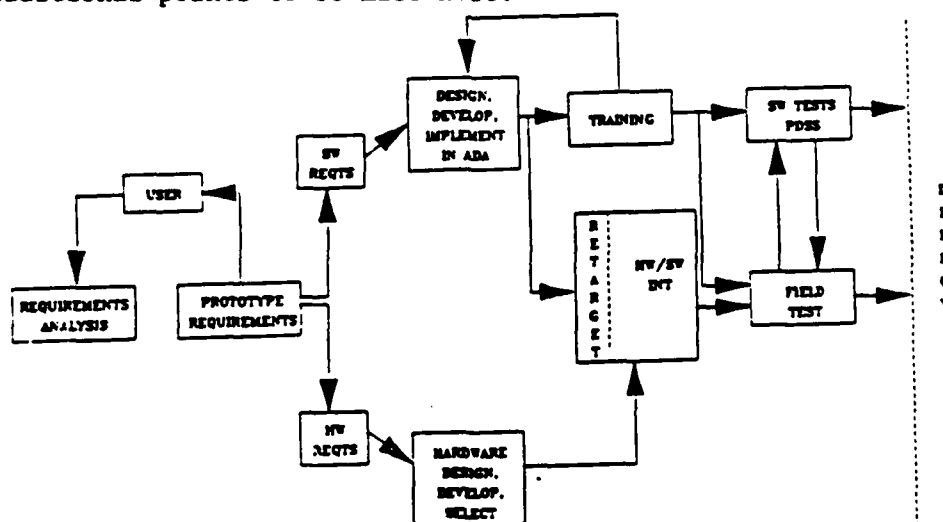


Figure 4-4. Interim Software First System Development Model

Once the requirements have been prototyped and the requirements definition has been completed, the requirements will be partitioned into software requirements and hardware requirements. The emphasis during the iterative loop in Figure 4-4 (the requirements analysis, prototype requirements, and the user boxes) will be on defining system requirements, without concern as to whether they will be implemented in hardware or software. Once these iterations have an adequately defined set of requirements, the requirements will be split into software or hardware requirements. There are several significant points here. The focus on system requirements is to assure that the system is being defined, not designed, during the requirements definition process. Secondly, the bias in this methodology is to classify requirements as software unless they can absolutely be implemented better in hardware. And, the high-risk requirements as well as the man-machine interface will be prototyped early in the design, develop, and implement phase. This prototyping process will help to further define the requirements, as will lessons learned during the training process. Therefore, the requirements for the system will be evolving in a controlled fashion, with the involvement of the user, after the initial requirements definition.

The retargeting of the software after development is intended to require a limited percentage of the development effort. During the software development, requirements for the hardware that will execute the software are being developed and assessed. Therefore, the selected target hardware will presumably meet the system needs, and will be compatible with the developed software. The integration portion of this box is in reference to integrating the hardware that implements hardware functions into the system.

Several elements that are critical to software first are implicit in Figure 4-4, but are not explicitly pictured. The continuous user involvement that was discussed earlier is not pictured. Although prototyping during requirements definition is pictured, the use of prototypes to explore high-risk requirements is not pictured. Also not pictured is the interaction between the software development and the

hardware selection. As mentioned earlier, the capabilities of existing hardware to execute the developed software will be monitored.

Prototyping during requirements definition also includes the development of the man-machine interface. The software first approach considers both the man-machine interface and the user to be integral parts of the complete system. The term interface is used here, although it is not to be interpreted as something added to the system, nor on the outside of it. The definition of the man-machine interface must begin during the system requirements definition process. Part of the system requirements definition must include a list of those functions that the human operator will control versus those that the system will control. This list will be refined through user feedback from prototyping and early training, as these techniques may reveal areas where the responsibilities initially assigned to the operator are beyond the capabilities of any one person to manage simultaneously, or that functions originally delegated to the system need human input. The man-machine interface will continue to be developed during the design, develop, and implement phase.

This emphasis on the user interface as an integral part of the system is an appropriate approach to any system development, not just for software first. Within the software first approach, however, the freedom from hardware constraints until much later in the development cycle allows more flexibility in the development of the user interface.

5.0 AREAS FOR SPECIAL CONSIDERATION

The software first approach to system development promises many advantages over traditional software development methodologies. Several areas of the model presented in Figure 4-4 must be further enhanced if these advantages are to be realized.

An initial area of improvement is system requirements definition. The intent of the requirements definition is to both assure concurrence between the user and the developer and to provide adequate articulation of the requirements to develop the "capability subsets" discussed earlier. This will require stating the requirements in a much more structured manner than is common in a standard development.

Another area investigated here is the area of software performance evaluation. Unfortunately, less research has been done in this area that is of specific interest to software first.

5.1 Requirements Definition

There are two principal rationale on which software first is based: Software is flexible and can be altered. If developed correctly, software can outline several generations of hardware. Each of these rationale must be qualified. Although software is flexible, the less it is forced to flex the more likely the software development and maintenance will be successful. To develop software correctly necessitates understanding user requirements and anticipating future requirements. Therefore, to both reduce the need for altering software and to increase the likelihood of long-lived software necessitates doing more thorough requirements analysis.

This section outlines the criteria for requirements analysis relative to supporting software first; Section 7.2 discusses the ability of current tools to satisfy these criteria.

Currently, requirements are typically written in a natural language. These documents vary in length, but are usually long enough to prohibit a detailed understanding of the system requirements by anyone other than the author of the document. This inability to understand requirements documents is because of the ambiguity and lack of specificity of natural language, and the inability to check for consistency or to maintain the document because of the lack of tools. In spite of these shortcomings, requirements continue to be written in natural language partially because reading natural documents requires no specific training. Because of these shortcomings, there is movement within the software engineering community toward more formalism in requirements documents.

To introduce formalism into requirements documents necessitates a representation scheme. Common representation schemes include finite state machines, program design languages, decision trees, and Petri Nets. The advantages of this introduction of formalism include the virtual elimination of ambiguity and the existence of tools to support the consistency and enhance the maintenance of the requirements document. The disadvantage to the introduction of formalism into requirements documents is that it necessitates training in the specific representation scheme. A discussion on specific techniques for introducing formalism into requirements documents is contained in Section 7.2.

At this time, no single technique for improving requirements documents is dominant. Several exist, each with its own advantages and disadvantages. What is necessary is a set of criterion to evaluate requirements techniques for a particular system development. Three key elements in the evaluation criteria are the user, the developer, and the application.

A set of eight criteria for evaluating a technique has been proposed [DAVI88]:

When the technique is properly used, the resulting document should be helpful and understandable to non-computer-oriented customers and users.

When the technique is properly used, the resulting document should be able to serve effectively as the basis for design and testing.

The technique should provide automated checks for ambiguity, incompleteness, and inconsistency.

The technique should encourage the requirements writer to think and write in terms of external product behavior, not internal product components.

The technique should help organize the document.

The technique should provide a basis for automated prototype generation.

The technique should provide a basis for automated system test generation.

The technique should be suitable to the particular application.

There are five additional criteria that must be added to this set to specifically support software first:

The technique should support iterative development of the document.

The technique should support stepwise refinement of the document.

The technique should support the identification of related and interdependent requirements listed in the document.

The technique should support the concept that the man, or operator, is part of the system.

Some elaboration of each of these criteria follows.

The technique must produce a document that is understandable to non-computer-oriented individuals because many system users are not computer-oriented. Many systems produce documents using a formal representation

scheme. Because the document must be understood by users who are not computer-oriented, the representation scheme must be easily understood. The need for some training is inevitable; a very short training session must be adequate or system users will not accept the requirements technique.

Once the requirements document is understood by both user and developer it must drive both the system design and the testing process. The requirements process must focus on what the system is to do and avoid addressing how the system is to do it. This line becomes blurred if, for example, the representation scheme is a PDL. This need to be the basis for a design must not become an excuse for allowing the requirements document to become a design document. Therefore, the system functionality must be evident in the design document without this document constraining the design. The statement that the requirements document should serve as a basis for design must be interpreted to mean that this document exposes the functionality of the system to the design team, not that it defines a particular design. The support for testing comes from the ability to easily interpret the functionality of the system into test cases. Again, the test cases are not to be embedded in the requirements document, but rather the requirements are to be stated in a manner that facilitates the development of test cases.

The primary shortcomings of using natural language for a requirements document are the ambiguity of natural language and the inability to formally determine properties such as completeness and consistency of documents written in natural language. Therefore, the primary capabilities for a technique other than natural language must include checks for ambiguity, incompleteness, and inconsistency. Given the size of the requirements documents of interest, these checks must be automated.

Underscoring the need to focus on requirements and avoid design during requirements analysis, is the criteria that the technique should encourage the requirements writer to think and write in terms of external product behavior. This emphasis focuses the requirements process on what the system will do and avoids the trap of defining how the system will do it. The

requirements writer must also consider that the system operator is internal, not external, to the system.

Requirements are frequently generated in a disorganized manner. The individuals involved in determining system requirements may generate a lengthy list of complex requirements. The technique must aid in structuring this list into a coherent document. More specific information on organizing the requirements document is provided with the criteria developed to specifically support software first.

Automated prototype generation and automated test case generation are two capabilities that are gradually becoming available in requirements techniques. These criteria would more accurately be stated as automated support for generation of prototypes and test cases than automated generation per se. The benefit of a formal representative scheme is the potential to automate the translation of requirements stated using the scheme into a prototype or test cases. The need for prototypes during requirements analysis, as stated elsewhere in this report, is to provide additional perspective on the requirements. The need for test case generation is to assure coverage of all requirements by the testing process.

Several of the existing requirements techniques were developed for a specific application domain. Some are applicable to other domains. Part of the evaluation criteria of requirements technique for a particular system is establishing the suitability of the technique for the specific application. The most direct measure of this suitability would be to establish that the technique had been successfully utilized by the user and the developer previously on a project in the same application domain.

The last four criteria were added to specifically support software first. The software first model proposes a strong commitment to iterative development of the system requirements. This translates into the need of a technique for developing a requirements document that support early and frequent changes to the document, and highlighting this ripple effect, are necessary capabilities. Also necessary are version control and archiving of

previous versions to enable the reconstruction of a previous iteration of the document if an implemented change is to be deleted. The requirements document in software first will evolve and this evolution must be supported by the technique used to develop the document.

Stepwise refinement is related to iterative development. The stepwise refinement process involves taking a requirement and refining that requirement by adding specificity and detail. This process does not change requirements but rather details the existing requirements. Support in this area includes tracing the evolved requirements, assuring that the refined requirements cover all aspects of the initial requirement, and identifying redundant elements in the set of refined requirements.

The organization and structuring of the requirements document noted earlier has specific implications for software first. Related requirements would include requirements that address a particular high-level function or capacity of the system. This criterion is to assure that the capability subsets discussed in Section 4.2 can be established. Interdependent requirements have correlational or casual links between them. These links must be identified by technique and exposed, on request, to the user.

Software first supports the concept that the system being developed includes the human with whom the system will be interacting. The implications of this concept include that requirements relative to the user must be included in the requirements document and that the man-machine interface is an integral component of the system, not an after-the-fact appendage. Assumptions about the capabilities of the human can be noted easily once the requirements relative to the human are in the requirements document. Automated support for extracting and itemizing these requirements is essential in deducing the capabilities the human is assumed to have. These assumed capabilities can then be compared to contrasted with the anticipated user requirements. Discrepancies can then be addressed as appropriate.

An assessment of the feasibility of these criteria given today's technology can be interpreted from this application to existing tools. This assessment is contained in Section 7.2.

5.2 Performance Evaluation

This section discusses issues of performance evaluation in the context of the proposed software first system development approach. In the software first approach, performance is first measured on the host prior to integrating the software on the target. Performance is then evaluated on the target as part of the final acceptance testing. The software first approach therefore requires performance measurement and evaluation techniques that can accurately predict the future performance of the software on the target from its performance in the host environment.

The software first approach also includes the use of performance evaluation techniques in the selection of target hardware. Comparisons of predicted performance on candidate targets will be used as discriminators. When new or upgraded hardware becomes available, performance predictions will be used to determine the merits of upgrading the system by porting to the new hardware.

The current state of the art in performance evaluation techniques is not sufficient to meet these software first goals. The emphasis of recent research in performance evaluation has been on the operational reliability of fielded software, rather than on prediction. As these techniques mature, they may be adaptable to a software first application. For example, some work has been done in performance estimation within software computer-aided design research. CAEDE, a graphical environment for structured design of Ada programs, can be used during the design phase of an Ada development to do the following analyses:

- o Workload consolidation
- o Bottleneck throughput limits
- o Analytic throughput and response time calculation
- o Simulation for throughputs and response times [WOOD86].

For real-time systems, it makes sense to do performance estimation earlier in the life cycle. As with test planning, early attention can uncover potential problems while they are cheaper and easier to correct. Performance issues may drive design decisions, and will influence implementation choices. For software first, early consideration of performance will feed into the hardware selection process, in concert with the database of hardware characteristics, and information on pending hardware improvements.

6.0 DETAILED DISCUSSION OF SELECTED TECHNICAL APPROACHES

6.1 Prototyping

This section discusses aspects of prototyping. The proposed approach to software first relies on prototyping to accomplish several goals. These include requirements definition, communication between user and developer, definition of the man-machine interface, training, and determining the feasibility of parts of the system.

6.1.1. The Motivation for Prototyping

Requirements are difficult to state explicitly and completely at the outset of a project. The user may know the need but be unclear about the details of the solution. Users and developers must deal with inherent communication boundaries. What one means to say may not be what the other hears. Waiting until later in the project to detect and correct a major misunderstanding is expensive in terms of money and time. The concept of prototyping encompasses a variety of specific techniques that facilitate communication and understanding between users and developers.

The man-machine interfaces of a system are particularly critical to the success of its development effort. They are complex and contain a wealth of detail. Prototyping is beneficial for determining detailed input and output requirements and man-machine interaction requirements. The use of prototypes can help identify and discriminate among which functions the user can effectively control and which functions the system needs to perform automatically. Prototypes of the man-machine interface can also be used for early training.

Some characteristics of system developments that influence the relative benefits of using prototyping techniques are the following:

- o Application area
- o Application complexity
- o Customer characteristics

o Project characteristics.

"In general, any application that creates dynamic visual displays, interacts heavily with a man user, or demands algorithms or combinatorial processing that must be developed in an evolutionary fashion is a candidate for prototyping" [PRES87]. Many CECOM systems display such characteristics and are therefore candidates for prototyping even if they are not following a software first approach. Prototyping is, however, critical to software first.

When building a system using new concepts or technologies Brooks [BROO75] contends

" . . . even the best planning is not so omniscient as to get it right the first time. The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers . . ."

Pressman contends that a prototype can serve as the throwaway system [PRES87]. This moves prototyping from a luxury if one has the time to an expected and required learning expense that can be recognized and minimized. The proposed software first system development approach echoes this conclusion.

The remainder of this section describes several kinds of prototyping and relates them to the proposed interim model of the software first development approach.

6.1.2. Paper Prototypes

Paper prototypes are images on paper of how the terminal screens, reports, physical interfaces, and actual system will look. Paper prototypes provide a useful and inexpensive way to improve communication during requirements definition. They are appropriate from the earliest stages of system definition. They allow the interface requirements to be determined

before any analysis or design is done. They result in solid requirements for the man-machine interfaces and in a reduced chance of misunderstanding-induced requirements changes later in the life cycle. Another consequence of using paper prototypes is that documentation of the man-machine interfaces, inputs, and outputs will be developed very early in the life cycle rather than after the system is built.

The software first approach to requirements definition stresses a need for the developer and user to understand each other and to agree on the system definition. The paper prototype approach fosters this interaction by requiring the developer to spend extra time with the user explaining the paper prototypes and listening to the user's comments.

6.1.3. Nonfunctional Prototypes

A nonfunctional prototype is software that accepts all valid inputs and displays a sample of each of the outputs, but has no functionality behind the displays. For instance, the software may accept a digitized image from an external source and display it on the screen. When the user requests a Fast Fourier Transform (FFT) and filtering of the image; however, another prestored image is displayed because the processing software to do the FFT and filtering functions has not yet been implemented.

Within a software first approach, nonfunctional prototypes provide the same advantages as do paper prototypes: improved communication and better requirements definition. The eventual users of a system can use nonfunctional prototypes to get a feel for the system being developed. If an appropriate man-machine interface is not obvious from discussion or paper prototypes, the developer can provide a few different nonfunctional prototypes that employ different man-machine interfaces. Feedback from the users can then help determine the proper approach. More importantly, the users can determine whether the man-machine interface maps to their conceptual model of the task to be accomplished.

Training early in the life cycle with user feedback to the developers is a key component of the proposed software first approach. This can be accomplished by using nonfunctional prototypes to provide the initial training for system users.

Techniques for building nonfunctional prototypes include the use of prototyping languages, 4th generation languages or conventional languages. The code developed for nonfunctional prototypes may be reused in the system, although the primary purpose of nonfunctional prototypes is to help define the system requirements, not to develop an initial implementation of the system.

6.1.4. Working Prototypes

Working prototypes have interface capability and limited functionality. Working prototypes are usually disposable. They are usually built with a 4th generation language or by exploratory programming. They are used to explore feasibility issues, such as whether or not a time constraint can be met.

Working prototypes are constructed from the high level design. Their applicability to the software first approach is in questions of feasibility. They may also be used to test design options. Working prototypes are generally meant to be used as a learning tool and then discarded.

6.1.5. Development Prototypes

A development prototype is a program that provides part of the desired functionality. The functionality is then augmented until the system is completed. This approach is also known as incremental development with user involvement. The software design must be available for the development prototype to be built.

Development prototypes may be used to establish requirements and then scrap the prototype. More commonly, however, they are used to provide a

continuous prototype for the user as the prototype is developed into the complete system. Such a continuously available, evolving prototype can provide a mechanism for user interaction throughout the development cycle, as emphasized in the ideal model of the software first approach, and implied in the interim model.

Prototyping has also been used to test various approaches to solving a problem. This technique is also known as exploratory programming. A developer who is unsure as how best to design or implement a new technology or deal with a new requirement may turn to prototyping to acquire some experience in the area before committing to a particular development path. For software first, this applies to exploring high risk requirements, especially those that affect the eventual selection of hardware for the system.

6.1.6. Prototyping Tools

The goal of prototyping is to learn the most about a problem for the least cost. This translates into wanting results without spending a lot of time coding. This issue has been addressed by recent developments in techniques for rapid prototyping. Rapid prototyping encourages the use of very high level languages and the reuse of code. Included in these categories are 4th generation languages, prototyping languages, demonstration languages, and object oriented development environments such as Smalltalk and LISP environments. The software first system development approach requires the use of such tools to achieve its goals of improved requirements definition, better communication between users and developers, more careful consideration of the man-machine interface, and early training, through the use of prototyping.

6.2 Calibrating Software On The Host Environment

To fully realize the benefits of developing systems using software first, and the benefits of performing this development on the rich host

environment, additional activities must be performed on the host environment. The host environment must be calibrated to "measure" the software and its needs as it executes. The amount of memory used during execution and the amount of processor time required for execution are two key factors to be measured and examined in detail. The intent of measuring the software is to identify the system's hardware needs. These needs include establishing the memory and speed requirements of the system hardware and determining the optimal system architecture.

In the traditional approach, the hardware portion of the system architecture is established and then the software is developed to ensure that the system meets its requirements. To change this process requires identifying specific, obtainable objectives for the process of calibrating the software while on the host environment. The specific issues relative to this calibration process, which will be discussed here, are simulating the target environment, identifying bottlenecks, and optimizing memory management. The final portion of this subsection establishes the objectives for this calibration process.

6.2.1 Simulating the Target Environment

Simulating the target environment requires evolving software first from its ideal phase to a more realistic position relative to the selection of hardware. The capabilities of existing hardware devices must be established and updated to determine the feasibility of system requirements. Also, simulating the target environment cannot be done without an understanding of potential system architectures. Therefore, simulating the target environment requires an understanding of current hardware capabilities and an understanding of potential hardware developments that may take place, or be required to take place, during the development phase of the system life cycle.

A database of hardware capabilities for an application domain is an integral part of software first. This database will include both the general purpose machines that are used in this application domain and the

special purpose processors that are developed or customized for these particular applications. The information on general purpose processors will not be domain-specific; the information on the special purpose processors will not only be more specific but also more extensive. This is because of the need for measuring different performance criteria, such as MIPS, bits per second, frames per second, or dots per inch.

An example from the image processing field illustrates this point. Image processing systems typically perform five processes. The first process is to receive digital data relative to a particular image or signal and process that data into a frame. This frame is further processed in the next step when a view of a region is developed. Once the region has been represented, processing of data relative to a specific object is performed. These three processes are managed by a control process. The final process is evaluating the image. No standard architecture exists for an image processing system. However, typically, the image/signal, region and object processes are performed on specialized processors, and the control and evaluation processing is performed by general purpose processors. To simulate the target environment for a signal processing system requires making some assumptions about hardware capabilities.

The information required on general purpose processors to simulate the target hardware architecture would not be as detailed as the information for the special purpose processors. This is because adding additional memory or computing power to a set of general purpose processors is fairly straightforward: add a memory board or go to a more powerful processor. The database must be informed of the most powerful processors. The information on the special purpose processors will be more exact. There are very few machines that perform the image, region, and object processing at the throughput that most image processing systems require. Therefore the exact specifications of this small set of machines can be maintained in the database.

In addition to the specifications for these special purpose processors, the development history and projections for near-term developments will also

be necessary. The state of the art in image processing hardware is being advanced continuously. One result of this continuing advancement is that the machines that will be available at the time a system is fielded will far exceed the capability of the machines available at the beginning of development. A primary tenet of software first is to take full advantage of this evolution. To do so requires both an understanding of the capabilities of special purpose processors for image processing and interaction with the hardware developers for the system. To simulate the target system requires simulating anticipated hardware. The accuracy and usefulness of the simulation is in part a function of the accuracy of this prediction of special purpose target hardware. This prediction will, therefore, not be left to chance.

We are proposing to write software drivers to simulate the hardware environment. The added cost of designing and developing the simulation code will be recouped in two ways. First, the correct hardware will be chosen which will reduce the development and maintenance costs of the system. Second, there is tremendous overlap between satisfying the objective of simulating the target hardware and satisfying the system requirements. What is being described here as an approach to simulating this target environment will also facilitate the integration of the software being developed and any hardware that must be concurrently developed.

Typically, special purpose image processors are developed as the key component of the hardware architectures for these systems. Because of the very demanding throughput rates, these processors have a set of primitive operations microcoded into either control memory or a programmable logic array. To simulate the target environment, these primitives must be established early in the development process. Once these primitives have been established, the hardware developers can produce the microcode needed to efficiently implement the primitives, and the software developers will be able to determine the basic operations of the special purpose processors.

Simulating these primitives will be possible, and the simulation should accurately reflect the functionality of the target hardware. Timing,

however, will have to be estimated because the simulation obviously will not approach the throughput of the target hardware. Also, a constant ratio of simulation time to real-time will probably not exist because the overhead of the simulation will not be consistent from primitive to primitive.

In this example, simulating a very complex target hardware environment is possible by controlling the development of the special purpose processors. The purpose of the simulation is to determine the eventual hardware needs of the system, to maintain the feasibility of those needs, and to determine potential system problem areas as early in the development cycle as possible so that the most effective solutions to these problems can be developed. Timing characteristics of the target hardware will be the most difficult to accurately simulate and measure. Because the goal of simulation is to support the calibration of the system software while executing on the host environment, and because timing characteristics will elude accurate assessments, another approach to complement these incomplete timing characteristics is presented.

6.2.2 Identifying Bottlenecks

Timing requirements are typically more critical than memory requirements because systems frequently have inflexible timing requirements based on specific needs and because, although it is always possible to buy more memory, the same is not true for more time. The speed of processing can be improved, but if this increase in speed involves increasing the number of processors, the management of multiple processors may degrade performance to the point where the anticipated increase in processor speed is not realized.

Because of the inexactness of any simulation of target hardware, the precision of timing needs based on calibrating the host environment will be suspect. Although these timing requirements for the target environment cannot be precisely established in a simulated environment, they can be adequately estimated to identify system bottlenecks. The objective of

identifying bottlenecks early in the development process is to allow time to objectively evaluate the alternatives to the parts of the system design that created the bottlenecks.

To continue with the image processing example, a segment of the system may be required to receive data, perform an FFT and then use this updated data to drive a graphics device. There are tradeoffs between performing the FFT in hardware and in software. The simulation of the target hardware can assume either choice. If the more flexible and maintainable software solution creates a bottleneck because of the slower processing time of the software implementation versus the faster hardware implementation, then the FFT may be done in hardware. If the simulation indicates that the output from the FFT will overwhelm the I/O channel to the projected graphics device, then alternatives must be explored to ensure that the information can be graphically presented.

In this example, the first decision may be whether to perform the FFT in existing special purpose hardware, or to highly tune the software to optimize efficiency, perhaps at the expense of some other software attribute such as modularity, portability, or understandability. The process of identifying bottlenecks worked in that it discovered a problem, based on a simulation of the target hardware environment and some assumptions about the system. In the second example, the graphics device and its I/O channel that were part of the simulated environment were determined to be inadequate for the system needs. Therefore, a hardware upgrade was necessary. The output of this is that either faster hardware is required, or system requirements must be revisited to ensure feasibility.

Another key component in identifying bottlenecks is the set of assumptions being made about the operating environment. Staying with the image processing example, assumptions about the data rates must be made to identify bottlenecks. These assumptions may involve peak rates of information flow, average rates of information flow, and precision requirements. The next example illustrates how these types of assumptions interact and how software first helps in clarifying both the assumptions and

the feasibility of requirements. The advantage of searching for bottlenecks in the simulated target environment is that it establishes the need to articulate these assumptions relative to the operational environment. Another alternative is now available in the event of a bottleneck.

If a system is to identify images and to respond to specific enemy images by firing a missile, requirements may be a function of data flow. The system may have requirements to launch two misses at an incoming threat. During the simulation to determine bottlenecks, it may be determined that at a full-scale attack, the missile launcher cannot fire fast enough to send two misses at each threat. Once software and hardware alternatives to increasing the speed of the missile launcher have been explored, it may be determined that the alternative is to fire only one missile per threat under maximum attack. Again, the intent is to identify bottlenecks, and to identify them early in the development cycle. In this example, the simulation of the target environment identified a bottleneck created by assuming that the system was being stressed to the maximum. The solution will be either enhancing the software or the hardware, or changing the requirement. The process was to simulate a target environment, make assumptions about the operational environment, and determine where the timing bottlenecks existed.

The objective of identifying bottlenecks is to assure that any shortcomings in the target environment are identified and corrected as early in the development process as possible. It is important to note that these shortcomings are being discovered on a simulated target environment. The advantage is that anticipated hardware has not been purchased and locked into the system; the disadvantage is that the bottlenecks have been identified on a simulation that is based on some assumptions. These assumptions will be based on moderately mature hardware and should be reliable.

6.2.3 Optimizing Memory Management

Calibrating the host environment involves measuring performance in the host environment to determine the needs of the target environment. The host environment is configured to facilitate the development of software and this configuration may be quite different from the target environment, which is configured to optimize system performance. A primary issue raised by this difference in configurations is that of memory management.

In a development environment, memory access is typically given to multiple programmers using the same computer. Typically, no user has a need for the entire system under development, or even a substantive portion of that system, in his address space. The memory management in the development environment is set to support several users doing relatively small pieces of a large system. This is inconsistent with the memory management to be utilized during the system's operation, when the memory management is set to optimize a single large program. The effect of this on the calibration process is twofold: units have a very different performance profile functioning independently than when they are functioning as elements of a large complex system, and tuning the operating system on the host environment to emulate the operational environment is a major task that will effect the development process.

Estimating the memory and timing requirements of a system requires executing the system as a system, not executing individual units of the system independently. This requires combining the units being developed by individual programmers into the system configuration, or into a large subsystem configuration. The memory management of the development environment may either prohibit this completely or require that large portions of the system or subsystem be placed into auxiliary memory. The degradation caused by page faults provides a misleading picture of the system's performance.

This issue of being unable to integrate a large section of the system in the host environment because of the memory management of the host is

common to software development efforts and is by no means a consequence of software first. It is, however, a major issue in software first because of the need to calibrate the software on the host environment. The system must be executed as a system to calibrate the software.

The other issue of memory management concerns tuning the operating system of the host environment to optimal performance for the system to function. Presumably the easiest transition from host to target is if the host is the target. Even in this scenario, the operating system must be tuned to support system execution, instead of system development. The changes are those noted earlier relative to memory management. The issue here is the amount of effort required to tune the operating system, coupled with the impact of this tuning on system development.

If the host is to be used for calibration, then the host environment must be used to execute the evolving system during development. This will likely cause restructuring the memory management of the host. The initial tuning of the memory management section of the operating system may require extensive experimentation; subsequent tunings will be much more straightforward. Scheduling these sessions of executing the evolving system can and should be planned to minimize their effect on the development process.

Again, the limitations of a host environment relative to its use as an execution environment exist without the software first approach. The fact that software first requires system integration early and on a continuing basis should be seen as a positive step. To realize the benefit, however, attention must be paid to these memory management changes and to planning for the adjustments to minimize any negative side effects.

6.2.4 Objectives of Calibration

At the highest level, the objective of calibrating the software as it is executed on the host environment is to measure certain attributes of the

software, notably memory and timing requirements. The measurement is done to define the needs of the target hardware.

This definition will not be done totally top down as portrayed by the ideal model. Knowledge of the existing hardware capabilities will be used to develop a rational first cut of the target environment. Recent hardware developments relative to the application domain will be used to project hardware capabilities that will exist at the point of system fielding. These projections will be used in defining the target environment to be simulated.

Once a simulated target environment has been developed, the performance of the evolving system software will be measured. This simulation may be partly hardware simulation, partly software simulation, and partly paper simulation. The objectives of executing the evolving system with this simulated target environment are threefold:

Refine the system architecture and make software versus hardware implementation decisions that are performance based.

Identify bottlenecks and problem areas early and determine how to avoid, or how to live with, the bottleneck.

Learn how to optimize the operating system and the tuning of the operating system to execute the evolving system on the host environment. This provide lessons learned on tuning the host operating system that will hopefully be transferable to the target environment.

6.3 Portability

Portability is a key ingredient of software first. It is both a driver of the philosophy and a goal of the development approach. The development of Ada as the first portable programming language, along with the recognition that software lifetimes should not be prematurely shortened by tying them to hardware lifetimes, makes software first both possible and appealing. The desire to have application software outlive its initial target hardware, or developing, training, calibrating and maintaining the

software on the host environment. transforms software portability from a nice ideal into a requirement.

This section discusses portability from definitions to implications, to suggestions for increasing it, and finally measuring it.

6.3.1 Definition of Portability

Software portability is the ease with which correctly functioning software running in an environment can be made to correctly function in another environment. An environment in this context means the hardware and operating system on which the application software runs.

Although software is often classified as portable or nonportable, in reality portability is a measure of the middle ground between two extremes. The smallest amount of effort to do a port is to recompile the software for the new target environment. The other extreme is to translate or rewrite the entire application while still retaining the original design. If the software application needs to be redesigned and rewritten, it is considered nonportable. Nonportable software must be redeveloped.

6.3.2 Benefits of Portability

Flexibility

Portability provides flexibility. It allows the customer to make use of preferred hardware and operating systems. It also allows the upgrade of hardware and software as dictated by changing needs. An environment upgrade could, for example, result in the ability to process the same amount of data faster.

Computer hardware technology has and is expected to continue to improve at a rapid rate resulting in:

- higher clock rates
- faster memory

new architectures (RISC, TRACE)
new devices (optical disks, OCRs, military unique devices)

In this climate, the environment chosen as the target at a project's inception may be obsolete by the time the system is fielded. Portability provides the flexibility to adopt ongoing technology advances. This flexibility allows for greater choice (and competition) in selecting hardware platforms.

Cost Savings

Portability provides cost savings, particularly in the maintenance phase of the software life cycle. Software with high portability will have few if any changes when a new release of an operating system or when a hardware upgrade is adopted.

A new class of hardware may become available with improved price performance. Porting the software to this new hardware will allow the system's performance to grow with hardware advances without incurring complete redevelopment costs. New software technology in compilers, development and maintenance tools, and operating systems will benefit portable systems. Taking advantage of hardware and software technology advances extends systems' useful lives. All these factors contribute to cost savings.

Time Savings

Portability saves time. The DOD is often looking for ways to reduce the length of the development cycle. Porting existing systems to new hardware and incrementally extending the systems capability is usually quicker than building a new system from scratch. This revitalization approach obviates some development efforts and provides an interim capability until new systems are fielded. In this way, portability can save time and complement the development cycle.

6.3.3 Challenges to Portability

Writing Portable Code

Portable software is not created by accident. To deliver highly portable software the developer must consciously address the following:

- language portability
- software developers' knowledge of machine dependencies
- documentation of environment dependencies and assumptions
- modular and parameterized dependencies.

Language compilers must be available for the target machines. The language semantics need to be unambiguously defined. The language implementation must not require uncommon hardware features. For portable applications it is beneficial to use a language that has a standard, such as Ada. Vendor unique language extensions decrease portability.

Programmers need software development experience to generate portable code. Programmers typically go through three levels of competence. First, they write the code so that it does what they want it to do. Second, they understand how it is that the code that they write instructs the machine to perform appropriately. Third, they understand the differences among target environments and write code that will do what is desired as independent of the environment as possible.

For example, seemingly innocuous statements can present portability problems:

```
if (A(x) and B(x)) then ... endif;
```

The interpretations may be:

If A(x) is FALSE then B(x) will not be executed

Either A(x) or B(x) may be executed first

If the first is FALSE the other will not be executed.

A(x) and B(x) will always be executed

In some applications the answers to these questions will effect the results and/or the performance of the application. Experience is important in writing portable code.

Dependencies and assumptions need to be documented. Documenting serves two purposes: It makes the developer more aware of the dependencies in the code, and it aids developers in porting the code in the future.

Assuming that environment dependencies will occur in the code, there still are steps that can be taken to keep portability high. One approach that works well is to localize the nonportable features in modules separate from the rest of the functional code. When the software is being ported, attention can be focused on the few modules that contain the dependencies rather than scanning and changing the code throughout the system. Encapsulation of hardware interfaces into modules also has the same benefit.

Another approach to increasing portability is to parameterize rather than hard code information. This is an accepted software engineering principle that directly affects portability. An example is where there is a location in memory that has information that the system accesses from many places in the code. Directly (hard) coding the address at each place it is referenced works, but portability would be increased if that address were defined as a constant in one place and the constant used everywhere else.

Meeting the System and Portability Requirements

Software portability is affected by the nature of the function performed by the software. Techniques to improve performance are often directly the opposite of portability recommendations. Embedded systems often have unique hardware interfaces for the devices with which they work.

Another tradeoff is developer "productivity" versus portability. If there is a software deadline, developers will spend less effort on portability in order to spend additional time to develop functionality.

This is because most customers give functionality higher priority than portability.

6.3.4 Approaches to Improve Portability

Robustness

Software that verifies inputs and results is easier to port. Robust coding aids in determining when assumptions are false in a new target environment. This can save debugging time when doing a port.

Standards

Use available standards when feasible. Defacto standards are also helpful. Use a well known approach rather than an equally effective home grown approach.

Understandability

The code should be easily understandable and maintainable by a less experienced person. Understandable code is valuable when the person who does the porting is different from the person who did the development.

Broadness

Use an approach that will work on the broadest class of machines. The "standard" may differ between two environments. To be portable, a different subroutine may have to be called to perform the same function.

Design

The design should be built such that it avoids depending on environmentally unique features. The design needs to modularize the environment dependencies. The designer should consider whether the unique

facilities can be emulated if the software is ported to another environment.

6.3.5 Issues

Performance Versus Portability

Experienced developers will tradeoff some efficiency for portability considerations because they know that it is the cost effective approach in the climate of ever faster/larger/cheaper hardware.

Portability can be improved by using only those language constructs and subroutine calls that are supported by all environments (all the environments on which the system is intended to run). This is known as common intersection or minimal support level. This means that the software has to run on the poorest environment or even a subset of the poorest environment.

For mundane programming, software can be made very portable. For embedded systems and/or time critical systems high portability is difficult or impossible. "When performance is of greatest concern, and a large number of calculations are required, it may be beneficial to use range constraints that translate exactly to common underlying hardware (16 or 32 bit) to allow the compiler to utilize hardware overflow detection during operations" [GRIE88]. The current approach with embedded systems is to get the most capability and performance from the hardware available. The developers will apply every trick possible to provide the most capable system with the hardware they have. The requirements for the system should specify where on this portability/performance spectrum the customer would like the system to be.

Long term solutions to the portability/performance tradeoff are continually being sought. One approach is to have military standard computers rather than a different architecture for each system. Another approach is for the compiler to deal with generating the most efficient code. Compilers continue to improve but in most cases are still not as good

as a human doing this. Ada is especially susceptible to this because Ada compilers are only about four years old.

Figure 6-1 depicts the hypothesized portability/performance curve. The line is theoretically the best that can be done. Systems to the left of the curve should be avoided because they can obtain additional portability without sacrificing performance.

In reality this graph may have many parallel curves or contour lines, where each line represents how much money the customer is willing to spend. Additional funding can buy some improvement and shift the curve to the right.

Real-time performance is an area where portability difficulties are obvious. Hardware speeds vary and what runs five seconds on a Cray 2 will take longer on an IBM PC. If the application is insensitive to or flexible on the time it takes to run, it is more portable. Timing quirks also occur, differences in device speeds can cause problems with servicing and missing interrupts or being interrupted at unexpected/inconvenient times. This can happen if the new target runs slower or faster than the original environment.

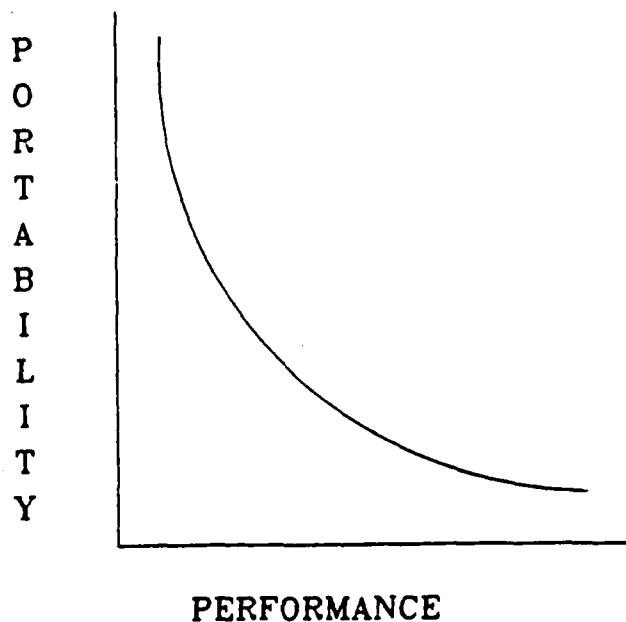


Figure 6-1. Portability/Performance Curve

6.3.6 Measuring Portability

Portability is a function of the software, the original environment, and the target environment. It is sometimes expressed as the amount of effort required to be able to port the software. This "amount of effort" metric is influenced by the knowledge, experience, and capability of the people performing the port.

The "amount of effort" metric can be used with the analogy approach. The first port of the system can be estimated using porting costs for similar systems and personnel. Later ports can be based on similar systems and the port history of this system. However, this approach cannot be used by the customer in evaluating the portability of the software delivered.

Several reports are available that address specific techniques to use and other techniques to avoid when writing portable Ada code [1815A], [GRIE88], [MATT87], [NISS82], [PAPP85].

6.4 Establishing Acceptance Criteria

6.4.1 Importance of the Acceptance Function to Software Development

Acceptance marks a key turning point in a system development. Acceptance is the process whereby the developer and the customer agree that the system is finished and meets the requirements that were set forth at the beginning of the development. The requirements to be satisfied encompass not only functional requirements, but also quality, performance, security, cost, schedule, and physical requirements for the hardware.

The acceptance process in traditional development approaches often turns out to be as much a test of the results of the requirements definition as of the results of the system development. This can cause grave problems, because it implies that the developer has finished a system

that is not exactly what the user/customer wanted. Of course, a lot can happen between requirements analysis and acceptance, so it is important to accept the system as conforming to the latest approved version of the set of requirements.

Assuming that the requirements are clear, consistent and complete, some of the technical subissues of acceptance are the following:

- o ways to demonstrate functionality
- o ways to measure performance
- o ways to quantify quality
- o ways to identify deficiencies
- o ways to resolve problems.

6.4.2 Traditional Approach to Acceptance

Software may be accepted as part of a system, or it may be accepted by itself. In either case, in the traditional system development approach, the acceptance test is done with the software already integrated onto the target hardware, and often in the target environment.

Another technique of the traditional approach is independent verification and validation (IV&V). This technique attempts to remove bias in the final testing by separating the testers from the developers. It is reasonable to have the IV&V contractor write the acceptance test plan, rather than the developer, based on an independent reading of the requirements. Otherwise, there is a risk of the testing being skewed toward the developers interpretation of the requirements or to focus on portions of the system that work well.

6.4.3 The Acceptance Function in the Software First Approach

In a software first development, one level of software acceptance will come before the software is ported to the target hardware. The judgement of the software can be made based on the quality of the software, and not on how the software conforms to hardware constraints.

The software acceptance may include recommendations for target hardware. The recommendations would be based on information in the database of hardware characteristics and the system performance requirements. A major difference will be that this level of acceptance testing of the software is completed on the host machine. One advantage of this is that the acceptance test suites will be available and will be able to be re-run on the host as the software is maintained, even in the face of target hardware changes.

A key technical issue for achieving software acceptance testing on the host involves being able to determine, with confidence, how the software will perform on the target. This issue was discussed in Section 6.2 of this report. In general terms, the software first approach will require that the robust host environment be instrumented, to assess eventual performance. Another major assessment that will need to be made involves the portability of the software. Both a measure of portability, such as how much effort will be required to port this software to its eventual target, and an assurance of portability, that the software will behave in the same way on the target, are needed. The portability guidelines being developed on a related task will be useful for both of these assessments.

Acceptance should be a continuing concern throughout the system development. The acceptance process is more than a single point in the development cycle. This is true for any software development approach. For software first, planning for acceptance will begin during the requirements iterations. The acceptance test plan will be developed at the same time as the requirements that the tests will validate are developed. This way the user has the same visibility into and influence over the final acceptance process as into the requirements definition. At this point tools and techniques can be employed to assure that the test plan addresses all of the requirements, to the satisfaction of both the user and developer. As the development continues, the test plan will evolve to reflect any changes in requirements.

6.4.5 Specific Challenges

One challenge that will be addressed during the balance of this effort is implementing the acceptance process within a software first development approach in a way that provides confidence to the customer that the software will perform acceptably in its eventual target environment. The software first approach will evolve to include evaluation criteria and measurement techniques that can be applied to the software on the host to predict its behavior on the target.

A related concern is to be able to measure the portability of the software on the host, so that an estimate of the resources required to port it to the target can be predicted with confidence. Portability was discussed in Section 6.3.

6.5 Graphically Oriented Techniques

This section discusses developments in visual programming and other graphical techniques. The proposed approach to software first will benefit from such techniques in improving communication between users and developers, and in developing the man-machine interface as an integral part of the system.

Graphical techniques are currently a popular research topic. This is partly because of the availability of powerful workstations with sophisticated graphics capabilities. Most of the impetus, however, comes from the demonstrated success of graphical techniques in other application areas, especially word-processing and data entry applications. The mouse and icon operating system interface of the Apple MacIntosh, for example, spawned many other such systems and variations. These graphically oriented systems are accessible to and understandable by non-technical users.

Visual programming encompasses the diverse research into bringing the success of graphical techniques in other application areas to software engineering. In many cases, the tools being developed provide automated

support for graphical techniques that software developers have previously done manually, for example data flow diagrams. The advantage is that the automated versions can include facilities for such things as consistency checking and traceability. Other tools that are being built reflect entirely new paradigms for software and system development, that could not be achieved by either manual or textual methods.

Alan MacDonald defined visual programming [MACD82]:

" . . . a methodology by which a person can direct a computer by showing it rather than telling it what to do. Visual programming represents a radical departure from the syntactical interfacing procedures that are common to traditional programming languages, and instead employs a visual interface between the user and the system. The user does not write a lengthy and exacting description of how his information should be formatted for entry into or retrieval from the system. Instead, he simply draws a visual representation of how he would like the information to be entered or retrieved."

The relationship between this view of visual programming and software first is that the users can define their needs using graphical means, and the developers can implement them directly. This eliminates the communication boundary between them because the requirements need not be translated into technical specifications (computerese) before they can be implemented.

The availability and use of graphically oriented tools and techniques also supports the software first prescription to do all development, testing and maintenance in the robust host environment rather than on the bare target machine.

Currently, graphically oriented tools are available for any part of the development life cycle, but they tend to focus on particular aspects. They are independent, and usually incompatible, thus it is not possible to link them together into a coherent development approach. As with the maturation of software tools in general, however, they will eventually be combined into graphical tool systems, or complete graphical development environments.

An example of a set of techniques for the graphical specification of large systems is PegaSys (Programming Environment for the Graphical Analysis of SYStems). PegaSYS supports the development and explanation of pictorial, non-textual descriptions of interactions among conceptual entities in a system design. The system includes rules and formalisms that detect invalid refinements of pictures [MORI84].

The design of large Ada systems is addressed by GODzilla, a graphic design assistant for Ada and information systems. A key feature of GODzilla is its ability to graphically specify both the procedural and data aspects of the system being designed. It assumes an object oriented design approach, and extends the standard notation for Ada packages, procedures and tasks. The designer manipulates these objects directly, and the system includes facilities for generating Ada code directly from the diagrams [POON88].

Other aspects of the software first approach, specifically development of the man-machine interface as an integral part of the system, user feedback during development, and early training of operators, are attainable through advances in graphically oriented techniques. The effect of graphics on user interfaces and training was explored in the development of "Steamer," an interactive simulation-based instructional system. The developers presumed that "graphical forms of representation provide powerful ways of bringing abstract things into the realm of the perceptually knowable . . . [because] people are especially good at: detecting patterns, constructing mental models of the world which support causal reasoning, and manipulating the world by actions on it or on representations of it" [HOLL86]. The project included the development of tools to assist in the creation of graphical interfaces that may be applicable elsewhere, including a model controller, a graphics editor, an icon editor, and several knowledge-based editors for specifying domain knowledge in the construction of the simulation.

7.0 USE OF COMPONENTS OF EXISTING METHODOLOGIES AND TOOLS

The software first system development methodology should make use of existing tools and components of other methodologies where appropriate. The principal goal of software first is to reduce total life cycle costs for major software-dependent DoD systems. This is the same goal as that of many recent and current methodologies. Therefore, IITRI has been reviewing other methodologies to determine which concepts, components, techniques, or tools already exist that can be utilized by software first. Some of these may be directly applicable, others will have to be adjusted. In either case, there are advantages to utilizing components of other methodologies.

The following advantages to reusing components from existing methodologies provide the rationale for taking this approach:

Most existing methodologies have already been 'proven.'

Experience with existing methodologies provides information on how well classes of problems are addressed.

Existing methodologies have been scrutinized by practitioners and the criticisms are available in the literature.

Complete methodologies take years to develop with the constant guidance of several people with broad experiences. This includes trial and error cycles using the fledgling methodology.

This section first discusses our investigations of existing methodologies with respect to the software first approach to system development. The remainder of this section presents information on software tools and discusses the degree of applicability that various tools and classes of tools have to the particular technical considerations.

7.1 Reusing Components of Existing Methodologies

An initial task in this research was to identify a "baseline" methodology, one that is fairly consistent with the software first approach, so that major components of the methodology could be directly utilized. A recent technical report [MAHA87] identified the major methodologies

currently in use. The 47 methodologies in the report were evaluated using the following criteria:

- o Consistent with the software first goals
- o Appropriate for the types of systems CECOM develops
- o Used often on defense systems.

These criteria are more completely defined below:

- o Elements consistent with software first goals
 - requirements definition
 - specifications
 - preliminary design
 - prototyping
 - detailed design
 - maintenance
 - traceability of requirements into design and code compatibility
- o Characteristics of CECOM systems
 - time-critical applications
 - embedded systems
 - device control
 - engineering applications
 - distributed processing
- o Used often on DoD systems
 - used by more than five organizations
 - more than six systems developed using the system

Of the 47 methodologies reviewed, five were identified as satisfying all of the above criteria:

- o Data Structured Systems Development (DSSD)
- o Object Oriented Development (OOD)
- o Problem State Language/Problem Statement Analyzer (PSL/PSA)
- o Statemate
- o Technology for the Automated Generation of Systems (TAGS).

These five methodologies were the prime candidates to become the backbone of the software first approach. In addition to looking at these five methodologies, other methodologies, techniques, and tools were examined. Additional contributing components were needed because no synthesis of the components of the first five produces a complete methodology that is congruent with the intent and goals of software first. The methodologies, techniques, and tools investigated are those that satisfy the specific software first needs.

Requirements definition is a key component of software first; therefore, much of the investigation into the literature focused on approaches to this task. In addition to identifying requirements that may not be implementable, the software first approach requires the identification of those requirements that are the most volatile, the most likely to change. These elements include compiler parameters, runtime parameters, and adaptable algorithms. The identification and segregation of these is necessary so that the design will be minimally affected by evolving requirements.

Once the requirements definition is complete, these requirements must be partitioned into software and hardware requirements. It is important to note that the requirements definition in software first is performed on system requirements to ensure that the system is being defined, not designed. Once the requirements are defined, a determination on each requirement's implementation, in software or hardware, must be made. The bias of software first is to implement requirements in software.

Several requirements are typically implemented in hardware and will be implemented in hardware using software first. These include sensors, radars, and communications equipment. The advantage to implementing these requirements in hardware is that the definition of the requirement exists and its feasibility is apparent. The disadvantage is that when a better piece of hardware is developed, there is an interest in updating the system hardware. This may require altering other components of the system that

interact with the hardware. Another disadvantage of implementing in hardware is that the hardware ages and may wear out.

An advantage of using Ada as the implementation language for a software first development is the package construct of Ada. If a particular requirement may be implemented in either hardware or software, and hardware may be the better choice if anticipated developments materialize, the decision may be postponed until later in the development cycle. An Ada package may be defined to satisfy the requirement. The package may be implemented in either software, hardware, or some combination. When the decision must be made to implement the package, the availability of hardware can be accurately assessed and the best decision made. This allows use of the most up-to-date hardware.

The literature review uncovered few approaches to determining if a requirement should be implemented in software or hardware.

The design phase of software first is not the same for all components of the system being developed. Certain high-risk requirements will be prototyped early, which will require early design. The man-machine interface will be designed and developed early. The use of capability subsets allows for portions of the system to be designed during the initial development but not implemented until a subsequent release. Each of these factors places an extra emphasis on design approaches that are flexible and conducive to requirements traceability. Also, the heavy emphasis on prototyping, and the desire to let the prototypes affect the design, require designs that contain not only the information on what a component is doing but on both how the component is doing its function and the intent of the design. The design techniques of prototyping-based methodologies were reviewed, but they do not completely satisfy the software first requirements.

This bottom-up portion of the development of software first employs the philosophy that it is better to utilize existing components than to develop everything from the ground up. The five identified methodologies are

consistent with the major goals and philosophy of software first. The proposed software first approach, however, requires both enhancing existing approaches and developing others to address specific tasks within its structure.

7.2 Tools

An important aspect of a development methodology is the selection and acquisition of tools to support the development effort. Tools should be available before the development effort begins as acquiring the tools and training personnel to use them can consume much time and effort. Software tools are an essential component of the total resources necessary to develop and maintain a system. Within a software first development approach, tools should be used both for traditional tasks and to help implement tasks that are unique to software first. For example, any software development, regardless of methodology, benefits from the use of code-oriented tools such as compilers, editors, linkers, loaders, and debuggers. The software first approach can benefit from the use of automated requirements analysis tools, although they may be used on system level requirements rather than strictly software requirements.

The remainder of this section presents results of our investigations into software engineering tools that may be useful when implementing a software first system development, with particular emphasis on their support for Ada and real-time software.

Methodology tools support project planning, requirements analysis, design, testing, configuration management, maintenance and other activities. Methodology tools are further classified as either method-independent or method-specific.

It is extremely important to choose tools that support the methods, but tools should not drive the methods. Most method-independent tools are vague and generalized and therefore inadequate. Most method-specific tools support functional decomposition methods such as Structured Analysis and

Structured Design. Relatively few tools support Ada-oriented software development methods. No production-quality tools support a majority of OOD steps. The system developer may well face the choice of a good tool for an obsolete method or an immature tool for a modern method.

Requirements analysis has been singled out as a key process in software first, worthy of new and better techniques for defining requirements. Therefore, existing tools for requirements analysis have been studied for applicability.

Automated tools for requirements analysis may be categorized in a number of different ways. Some tools have been designed to automate the generation and maintenance of what was originally a manual method and these tools typically make use of a graphical notation for analysis. This class of tools produces diagrams, aids in problem partitioning, maintains a hierarchy of information about the system, and applies heuristics to uncover problems with the specification. More importantly, such tools enable the analyst to update information and track the connections between new and existing representations of the system. For example, a number of Computer Aided Software Engineering (CASE) tools enable the analyst to generate data flow diagrams and a data dictionary and maintain these in a database that can be analyzed for correctness, consistency, and completeness. In fact, the true benefit of this, and of most automated requirements tools, is in the "intelligent processing" that the tool applies to the problem specification.

Another class of automated requirements analysis tools makes use of a special notation (in most cases this is a requirements specification language) that is processed in an automated manner. Requirements are described with a specification language that combines keyword indicators with a natural language narrative. The specification language is fed to a processor that produces a requirements specification and, more importantly, a set of diagnostic reports about the consistency and organization of the specification.

After the system-level requirements have been defined through user-developer interaction and the use of requirements analysis tools, the next step in the software first approach is to partition the requirements into software and hardware requirements.

Thanks to recent strides in microprocessor technology and in software engineering (e.g., the development of Ada), the size and complexity of embedded real-time systems have grown explosively. It is now practical to implement many functions in software that earlier would have been relegated to hardware. This flexibility has led to the software first bias toward software implementation of a function whenever possible. When the project begins, the designers do not know the functional system division between the hardware and the software. Tools that supply the hierarchical functional decomposition framework can be used to help define exactly which functions should be performed in hardware and which in software.

Several of the tools that implement the common methods for requirements definition and requirements analysis have recently been evaluated against several criteria that are defined in Section 5.1 [DAVI88]: output that is understandable to non-computer-oriented users; output that forms a basis for design and testing; automated checks for ambiguity, incompleteness, and inconsistency; system view is in terms of external behavior; output should be organized; tool should support automated generation of prototypes and test cases; and the tool should support the specific application. The tools are based on either finite state machines, decision tables or decision trees, program design language, structured analysis, or Petri nets.

Finite state machines appear to provide the superior representation against the stated criteria, with the only major drawback being the amount of training to understand the tools and their inputs and outputs. Decision tables or decision trees are most appropriate for decision-intensive applications, and tools based on this model do not provide for automated checking of requirements for ambiguity, incompleteness, or inconsistency. Further, these tools do not provide support for prototype or test case generation. The strength of program design language tools is that they are

intuitive, because of their similarity to natural language; the weakness is the lack of formalism necessary to assure well structured documents to drive well structured designs. Static analysis can be performed to detect structural errors, but this is more beneficial in the design phases than in requirements definition. Structured analysis tools, even those extended to support real-time applications, are more appropriate for systems based on data flow and data structure. This leads to a data-flow view of the system, not an external view. Static structural and behavioral analysis can be performed, but the structure is data driven. Petri nets are strong at representing synchronous behavior, but appear to be hard to master, particularly for the non-computer-oriented user.

Tools based on the finite state machine model appear to be the most supportive of requirements definition process detailed in Section 5.1.

The next major class of methodology tasks of interest are those that assist in designing software that satisfies requirements.

PDL has traditionally referred to a pidgin language that uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e. a structured programming language). More recently, PDL has also come to mean Process Definition (Description) Language and Program Documentation Language.

The benefits of a language-specific PDL, such as Ada PDL, are many. The power of Ada can be used in the design process to use Ada's features to achieve the goals of modern design methods; specify unit interfaces in a compiler-enforceable manner and specify the design of data as well as algorithms. A language specific PDL eases the transition from design to code and ensures consistency by evolving the PDL into deliverable source code. A PDL also enhances communication among developers.

One problem with PDLs is they are linear in nature, whereas the high level software architecture has a graph structure. This is why graphics are superior to PDLs for showing the preliminary design in terms of programming

units and their relationships. Another problem with PDLs is that they may encourage premature implementation decisions. Finally, in the case of Ada PDL, program managers, systems engineers, and hardware developers who are unfamiliar with both Ada and PDLs may find it difficult to review.

When choosing a PDL, it is best to choose a compilable PDL so that project tools may be used to ensure automated support for syntax and interface checking, design debugging, set-used listings, etc. In this way the developer/analyst ensures that the PDL is compatible and integrated with the project development method(s) and tools.

Many of the method-specific design tools available today support structured analysis/structured design and many have incorporated the rules and icons needed for real-time software analysis and design methods. These tools, however, do not address all the aspects of designing Ada code for embedded systems.

While functional and data flow specifications usually suffice to characterize software, the concurrent nature of real-time embedded systems demands more information. Designers must decide whether modules are tasks, interrupt service routines, device servers, or synchronous program units. Intertask synchronization or communication mechanisms (queues, mailboxes) must enhance data flow descriptions to insure the integrity of the data being passed. Control signals from hardware (like interrupts) and software (like events or semaphores) must also be added.

There are many CASE (Computer Aided Software Engineering) tools currently on the market that provide automated support for Structured Analysis/Structured Design (SA/SD). There are significantly fewer tools that address a majority of the concerns of real-time systems. SA/SD tools provide support for drawing data flow diagrams as a basic building block of the tool. Some have begun to address the concerns of real-time systems by adding control flow information to the data flow diagrams and by adding the capability to draw state transition diagrams.

These tools typically perform global analysis over the entire system model to verify consistency and completeness. They also perform automatic report documentation, in some cases satisfying DoD-STD-2167A requirements. Once the analyst has designed the system, these design tools will automatically generate a variety of charts depicting module hierarchy, the functional network, and a graphic representation of the system structure. Some tools will go further and perform an automatic transformation from analysis, thus ensuring that the integrity of the analytical model is carried over into design.

The transformations attempt to enforce information hiding and precise interface definitions. The result of the transformation is a suggested system partitioning into modules, sub-systems and functions. The designer is free, of course, to add, delete, merge or split the modules as required to fully define the system. Finally, a majority of the tools provide some measure of traceability; automatically tracing items identified in the requirements analysis phase through the architectural and detail design phases. Traceability helps to ensure that each requirement is satisfied in the code and that each block of code maps to one or more requirements.

Much less prevalent today, are design tools dedicated to the development of real-time embedded systems. Such tools go beyond adding control information to data flow diagrams and attempt to tackle the toughest part of defining real-time systems; specifying timing relationships with the outside world. Computer Aided Real-Time Design (CARD) technology supplies both the basic data-flow and hierarchical views and many additional details unique to real-time systems.

In addition to the traditional architectural model, real-time systems also need further specification via a tasking model and an exception (interrupt) model. CARD tools from Ready Systems is an example of a tool that more completely supports the analysis and design of real-time systems. It supports data-flow and control-flow diagram editing and data flow analysis. It provides graphic design tools for developing the multitasking model. CARD tools also provides real-time performance verification

utilizing manufacturer's processor and memory specifications to calculate I/O overhead and synchronization/communication overhead.

The actual process of coding in a software first development may not be very different from a traditional coding effort. The key characteristics of coding for software first are that the implementation language will be Ada, and that all practical guidelines for increasing portability will be employed. Therefore, code-oriented tools that are useful for traditional software development should be useful for software first.

In the category of code-oriented tools, the debugger is the single most important tool after the compiler. To be most effective, it is important to have windows showing source, input and output. A good debugger allows the coder to step through the source, show the call hierarchy, set break points, deposit and examine data values and handle large collections of units without exceeding virtual memory. An Ada-oriented debugger should additionally show the state of tasks and locate overloaded objects (i.e., determine where they are defined). The debugger should use the designer's identifiers, even when they are very long, rather than generating incomprehensible arbitrary identifiers of its own. Finally, the debugger should handle all valid code (e.g., deep nesting and Ada separates and library units) without getting lost.

Another potentially useful code-oriented tool is the language-sensitive editor (LSE). LSEs prevent syntax errors, promote the learning of the language and are generally very useful for beginning programmers. They may, however, be too slow for advanced programmers. LSEs usually take one of two approaches, the template or the parsing approach.

In the template approach, the user typically chooses a construct from a menu or series of menus, the LSE generates a template with all required components and the coder fills in the details. At some point the developer becomes familiar enough with the language that the menus get in the way and actually impede progress. The parsing approach allows the developer to directly enter the code that is then parsed by the LSE for syntactical

errors. LSEs should be integrated with method-specific tools that automatically generate PDL and they should automatically produce skeleton prologues.

8.0 REFERENCES

- [1815A] Ada Programming Language, ANSI-MIL-STD-1815A, Department of Defense, January 1983.
- [AGRE86] W.W. Agresti, "SEL Ada Experiment: Status and Design Experiences", Proceedings of the 11th Annual Software Engineering Workshop, NASA/GSFC, December 1986.
- [BOAR84] B. Boar, Application Prototyping, Wiley-Interscience, 1984.
- [BOEH73] Boehm, Barry. "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5, May 1973.
- [BOEH87] Boehm, Barry. "Improving Software Productivity," Computer, Vol. 20, No. 9, September 1987.
- [BOOC83] Booch, Grady. Software Engineering with Ada, Benjamin Cummings Publishing Company, Menlo Park, CA, 1983.
- [BOOC87] Booch, Grady. Software Components with Ada, Benjamin Cummings Publishing Company, Menlo Park, CA, 1987.
- [BRIC87] D. Bricklin, DEMO II, (software), Software Garden, 1987.
- [BROO75] F. P. Brooks Jr., The Mythical Man-Month, Addison-Wesley, 1975.
- [BROO87] Brooks, Frederick. "No Silver Bullet - Essence and Accidents of Software Engineering," Computer, Vol. 20, No. 4, April 1987.
- [BROO88] Brooks, Craig, Edward J. Gallagher, Jr., and David Preston. "The Software First System Development Methodology," Proceedings of the 6th National Conference on Ada Technology, Crystal City, VA, March 1988.
- [CAST87] Castor, Virginia and David Preston. "Programmers Produce More With Ada," Defense Electronics, June 1987.
- [DAVI88] Davis, Alan M. "A Comparison of Techniques for the Specification of External System Behavior," Communications of the ACM, Vol. 31, No. 9, September 1988.
- [DEBA86] De Bartolo, Gil and Ron Richards. "The Back-End of a Multi-Target Compiler," Proceedings of the 4th National Conference on Ada Technology, March 19-20, 1986.
- [FALK88] Falk, H. "CASE Tools Emerge to Handle Real-Time Systems," Computer Design, January 1, 1988.
- [FELS88] R. C. Felsing, Object Oriented Design, seminar notes, 1988.
- [FLEI76] Fleisher, Robert J. "Software-First System Design," Compcon76, February 24-26, 1976.

- [FREE83] Freeman, Peter and Anthony Wasserman. Software Design Techniques, IEEE Computer Society Press, Los Angeles, CA, 1983.
- [GOLU82] Golubjatnikov, Ole. "Architecture, Hardware and Software Issues in Fielding the Next Generation DoD Processors," Proceedings of the 2nd AFSC Standardization Conference, December 1982.
- [GRIE88] L.J. Griest and T.E. Griest, Preliminary Transportability Guideline for Ada Real-time Software, LABTEK Corporation, Woodbridge, CT 06525, April 30, 1988.
- [HOLL86] Hollan, J.D., E.L. Hutchins, T.P. McCandless, M. Rosenstein, and L. Weitzman, Graphical Interfaces for Simulation, Institute for Cognitive Science Report 8603, University of California, May, 1986.
- [IITR88] IIT Research Institute, "Establish and Evaluate Ada Runtime Features of Interest for Real Time Systems - Interim Report", Contract MDA 903-87-D-0056, U.S. Army CECOM, March 1988.
- [KRUC84] P. Kruchten, E. Schonberg, and J. Schwartz, "Software Prototyping Using the SETL Programming Language", Software, Vol. 1, NO.5, October 1984.
- [MACD82] MacDonald, Alan, "Visual Programming," Datamation, Vol. 28, No. 11, October, 1982.
- [MAHA87] Mahajan, L., M. Ginsberg, R. Pirchner, and R. Guilfoyle, "Software Methodology Catalog," Technical Report MC87-COMM-ADP-0036, U.S. Army Communications-Electronics Command, October, 1987.
- [MANT88] M.M. Mantel and T.J. Teorey, "Cost/Benefit for Incorporating Human Factors in the Software Lifecycle", CACM, Vol. 31, no. 4, 1988.
- [MATT87] E.R. Matthews, "Observations on the Portability of Ada I/O", ACM Ada Letters, Vol. VII, no.5, 1987.
- [MCFA88] G. McFarland, P. Brennan, J.D. Litke, M.S. Restivo, "A Tool Set for Distributed Ada Programming", Grumman Data Systems, Woodbury, NY, 1988.
- [MCGA88] F.E. McGarry, W.W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)", Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988.
- [MORA78] Moralee, Dennis. "MIL-SPEC Computers - Building the Hardware to Fit the Software," Electronics and Power, August 1978.
- [MORI84] Moriconi, Mark, "Representation and Refinement of Visual Specifications in PEGASYS", RADC-TR-84-128, June 1984.

[NASA87] National Aeronautics and Space Administration, Goddard Space Flight Center, "Ada Runtime Environment Issues", preliminary report, June 1987.

[NIEL88] K.W. Nielsen and K. Shumate, "Designing Large Real Time Systems with Ada", CACM, Vol. 30, no. 8, 1987.

[NISS82] Nissan, Wallis, Wichmann, and others, "Ada-Europe Guidelines for the Portability of Ada Programs", ACM Ada Letters, Vol. 1, no.3, 1982.

[PAPP85] F. PAPPAS, Ada Portability Guidelines, SofTech Inc., Waltham, MA, March 1985, DTIC/NTIS #AD-A160 390.

[PRES87] R.S. Pressman, Software Engineering - A Practitioner's Approach, second edition, New York, NY., McGraw-Hill, Inc., 1987.

[POON88] Poonen, G., H.M. Nachiappan, S.O. Landstrom, "A Graphic Design Assistant for Ada and Information Systems," Proceedings, 6th National Conference on Ada Technology, 1988.

[SCH185] J. Schill, R. Smeaton, R. Jackman, "The Conversion of Commands & Control Software to Ada: Experiences and Lessons Learned", Ada Letters, Vol. IV, Issue 4, 1985.

[SIVL87] Sivley, Karen E. "Experience and Lessons Learned in Transporting Ada Software," Proceedings of the Joint Ada Conference Fifth National Conference on Ada Technology and Washington Ada Symposium, March 16-19, 1987

[TAFT88] Taft, Darryl K., "Lab Supporting Development of Ada Tasking Tools", Government Computer News, April 15, 1988.

[TATE85] G. Tate and T. Docker, "A Rapid Prototyping System Based on Data Flow Principles", ACM SIGSOFT Software Engineering Notes, Vol. 10, no. 2, April 1985.

[WIEN84] R. Wiener and R. Sincovec, Software Engineering with Modula-2 and Ada, New York, NY., John Wiley & Sons, Inc., 1984.

[WILL87] Williams, T. "Real-Time Development Tools Aid Embedded Control System Design," Computer Design, October 1, 1987.

[WOOD86] Woodside, C.M., "The CAEDE Performance Estimator for Structural Designs of Ada Programs," 9th Minnowbrook Workshop on Software Performance Evaluation, August 1986.